TU WIEN Informatics

Seamlessly Interfacing Automation Systems with Simulation Environments

A Case Study for FMI and OPC UA

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Andrej Kurtović
Matrikelnummer 01428987

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Univ.Ass. Dipl.-Ing. Dipl.-Ing. Dr.techn. Gernot Steindl, BSc

Wien, 3. Dezember 2021

Andrej Kurtović	Wolfgang Kastner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Seamlessly Interfacing Automation Systems with Simulation Environments

A Case Study for FMI and OPC UA

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Andrej Kurtović
Registration Number 01428987

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Univ.Ass. Dipl.-Ing. Dipl.-Ing. Dr.techn. Gernot Steindl, BSc

Vienna, 3rd December, 2021

Andrej Kurtović	Wolfgang Kastner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Andrej Kurtović

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Dezember 2021

Andrej Kurtović

Acknowledgements

I would like to thank my advisor, Prof. Wolfgang Kastner, for the support and suggestions, as well as for giving me the possibility to work on the topic of this master thesis. I would like to thank my co-advisor Univ.Ass. Gernot Steindl for the suggestions, support, and many constructive comments and ideas that have made this thesis possible.

I would also like to thank my mother, and the rest of my family for supporting me throughout my studies.

Kurzfassung

Die Digitalisierung der Industrie und die damit verbundene vierte industrielle Revolution (Industrie 4.0) führen zu einem zunehmenden Grad an Automatisierung, sowie automatischen Datenaustausch durch Maschine-zu-Maschine Kommunikation. Die daraus resultierenden Cyber-Physische Systeme (CPS) bieten durch die Vernetzung der Maschinen neue Funktionalität, und ermöglichen die Zusammenarbeit der digitalen Welt mit der realen Welt. Ein wichtiger Bestandteil von CPS sind Simulationen, die sowohl während des Betriebs (zum Beispiel als Teil des digitalen Zwillings) als auch während des Systementwurfs (zum Beispiel als Hardware-in-the-Loop) verwendet werden können. Ein häufig auftretendes Problem, das bei der Kopplung zwischen dem Automatisierungssystem und der Simulationsumgebung eines CPS auftritt, ist, dass diese Systeme nicht mit Blick auf die Interoperabilität (untereinander) konzipiert wurden.
Aus dieser Problematik ergibt sich die folgende Forschungsfrage: *Wie sieht ein geeigneter Arbeitsablauf aus, um ein Automatisierungssystem, welches durch ein Informationsmodell beschrieben wird, (semi-)automatisch mit einem bestehenden Simulationsmodell zu verbinden, so dass das Automatisierungssystem während der Laufzeit auf die Simulationsdaten zugreifen kann?* Ein solcher Arbeitsablauf muss eine Interoperabilitätsschicht (eine sogenannte Mapping Einheit) enthalten, um die beiden Systeme zu verbinden und die Kommunikation zwischen den beiden Systemen zu ermöglichen.
Um die Forschungsfrage zu beantworten, wurde folgender methodischer Ansatz gewählt: Zuerst wurde eine Literaturrecherche durchgeführt, um einen Einblick in den Stand der Technik sowie die Möglichkeiten zur Gestaltung einer solchen Interoperabilitätsschicht zu erhalten. Dann wurden mögliche technische Arbeitsabläufe untersucht. Aus der Analyse ergaben sich einige Anforderungen an das Design der Mapping Einheit. Anschließend erfolgte eine Evaluierung mithilfe einer Proof-of-Concept-Implementierung, um die Machbarkeit des vorgeschlagenen Arbeitslaufes und des gewählten Ansatzes zu bewerten.
Unter Verwendung der offenen Industriestandards FMI und OPC UA wurde anhand der Proof-of-Concept-Implementierung gezeigt, dass eine Mapping-Einheit entworfen werden kann, die halbautomatisch die Schnittstellen von Automatisierungssystemen und Simulationsumgebungen verbindet und den Datenaustausch zwischen ihnen ermöglicht. Die Funktionalität der Proof-of-Concept-Implementierung wurde an zwei Anwendungsfällen mit unterschiedlichen Schwerpunkten evaluiert. Dabei wurde sowohl die halbautomatischen Matching-Fähigkeiten als auch die Laufzeit-Datenanbindung zwischen den Systemen gezeigt und deren aktuell bestehenden Einschränkungen diskutiert.

Abstract

The digitalization of the industry and the associated fourth industrial revolution (Industry 4.0) leads to increased automation, as well as to automatic data exchange through machine-to-machine communication. The consequent Cyber-physical systems (CPS) offer new functionalities through machine interconnection, and enable cooperation of the digital "cyber" world with the "physical" world (the real systems). An important part of CPS are simulations which can be used during operation (for example as a part of the digital twin), as well as during system design (for example as hardware-in-the-loop). A problem that often arises when connecting automation systems and simulation environments of a CPS, is that the systems were not designed with interoperability (with each other) in mind.

This problem results in the following research question: *What is an appropriate workflow to (semi-)automatically connect an automation system described by an information model to an existing simulation model - enabling the automation system to access the simulation data during runtime?* Such workflow needs to incorporate an interoperability layer (a so-called mapping unit), in order to connect the two systems, and to enable the communication between them.

In order to answer the research question and achieve the goals of this thesis, the following methodological approach has been taken: first, a literature survey has been conducted, in order to gain insight into the state of the art, as well as the possibilities for designing such an interoperability layer. Next, possible engineering workflows were explored. The analysis yielded some requirements on the design of the mapping unit. And lastly, a proof-of-concept implementation was designed, which was used to evaluate the feasibility of the proposed workflow and the chosen approach.

Utilizing the open industry standards FMI and OPC UA with the proof-of-concept implementation, it was shown that a mapping unit, which semi-automatically connects the interfaces of automation systems and simulation environments and enables data exchange between them, can be designed. The functionality of the proof-of-concept implementation was evaluated using two use cases with different focuses, thus demonstrating both the semi-automatic matching capabilities, in addition to the runtime data flow capabilities between the systems, as well as the current limitations of the proof-of-concept implementation.

Contents

CHAPTER 1

Introduction

Motivation

The ongoing fourth industrial revolution - Industry 4.0 - brings more automation, integration, and machine-to-machine communication. The goals of Industry 4.0 are higher operation efficiency, and productivity, as well as higher levels of automation [TS16]. An important part of modern production systems are simulations, which are utilized within the cyber-physical systems (CPS). CPS are industrial automation systems which incorporate new functionalities through machine interconnection and communication, in order to enable the cooperation of real physical systems, with the digital world [Lu17]. In particular, major features of Industry 4.0, such as digitalization, optimization, customization of production, automation and adaptation, human-machine interaction, and automatic data exchange and communication [RMK16], could benefit from the effective utilization of simulations.

Simulations can be utilized during the operation of the cyber-physical system, for example as a part of a digital twin, to monitor, diagnose, predict, control, and reconfigure parts of the system [SSK+20]. In order for the simulations to produce reliable outputs, the simulation models need to be adequate representations of reality. To this end, simulation verification and simulation validation should be used, in order to ensure the accurate and adequate representation of reality, by the simulation. [Led99]

On the other hand, it is also possible to utilize the simulations during system design, to verify the system's functionality. This is usually achieved by hardware-in-the-loop, meaning that parts of the tested systems are simulated, while other parts are already implemented in real hardware. Hardware-in-the-loop simulation requires that simulation models for some parts of the system are developed, including all significant interactions with its operational environment [Bac05]. The simulation then monitors the output signals of the system under test, and generates input signals for it, at specific time points. [Led99] The advantage of this approach is that the finished system components

can already be tested, even while other system components are still in development. The issue that arises with this approach is that the simulation models are often developed completely independently from the existing automation systems, and thus there are interoperability problems when the simulation has to interact with the real part of the automation system. In addition, the process of converting the programs from the simulation environment to the deployment environment often introduces errors, and prolongs time-to-deployment of the system.

An attempt to alleviate this issue is proposed in this thesis, using a mapping unit which should seamlessly connect the simulation environment and the rest of the automation system. In a sense, an interoperability layer, called the mapping unit, is introduced between the simulation system and the rest of the automation system. This interoperability layer is tasked with bringing the two interfaces together, seamlessly. The interfaces of these systems usually consist of different typed variables complete with additional information, such as engineering units, minimal, and maximal values. The name mapping unit comes from its task: mapping variables of one system to the variables of the other system, as well as connecting the mapped variables together enabling data flow during execution. In order to achieve this, it needs to take the additional provided information into account, and create a correspondence map between pairs of variables of the two systems.

Problem Statement

Ideally, the simulated components could be added and removed from the automation system at will, either during system development to check the system's interaction with unfinished components, or during deployment, in parallel with the working system, to provide additional insight into the workings of the real system. In reality, the interfacing usually presents an issue, and prevents such seamless interaction. The interfaces are usually not directly compatible, and making the interaction between the systems function properly is a time-costly task. Additionally, errors might be introduced, and these may be difficult to find, especially during system design when not all components are verified as correct. The proposed interoperability layer - the mapping unit would alleviate these issues, by (semi-)automatically finding the mapping between the interface connection points, as well as seamlessly connect the two interfaces, facilitating for communication between them.

This problem statement yields the following research question: *What is an appropriate workflow to (semi-)automatically connect an automation system described by an information model to an existing simulation model - enabling the automation system to access the simulation data during runtime?*

Methodology

The workflow needs to incorporate an interoperability layer, or a mapping unit, which would connect the two systems. The requirements for such a mapping unit, which would connect the automation system with the simulation system would be as follows:

- First, the mapping and the connection should happen automatically on one hand, but also error free on the other. Complete automation and especially the property of being error free might be very difficult to achieve.
- The interoperability layer should lean into safety, and thus only connect the points which are determined to be compatible with a determined degree of certainty.
- Additionally, the interoperability layer should provide a way for human to intervene, and define, re-define, and remove the connection points.

In order to create such an interoperability layer, the following workflow is expected: first, the descriptions of the interfaces of the simulation model, and the automation system model should be obtained, if available. Then, based on these descriptions (or alternatively the proposed descriptions, if real descriptions do not exist yet), an internal mapping of the connection points between the two interfaces should be created. Lastly, the communication between the simulation and the automation system should be enabled, that can either work during system operation, or be used during testing.

In order to achieve these goals, the following methodological approach will be taken: first, a literature review will be conducted, which will provide insight into the current state-of-the-art, as well as different approaches, possibilities, and methods which can be used to design such an interoperability layer. The literature should also provide a theoretical insight into the problem. Second, the possible different variations of the problem, and the accompanying approaches should be explored, in order to obtain insight into the possible issues and solutions that might arise from different specifications. Next, the mapping techniques will be analyzed to find a way in which to map the connection points (variables) of the two interfaces. Different possible criteria, as well as the mapping certainty should be discussed and explored. Lastly, a proof-of-concept implementation should be developed, utilizing modern frameworks, in order to evaluate the feasibility of the proposed workflow, as well as the proposed mapping techniques.

Technology

For the proof-of-concept of this thesis, the following two frameworks were chosen: the Functional Mockup Interface (FMI) for the simulation side, and the Open Platform Communications Unified Architecture (OPC UA) for the automation side of the system. These two frameworks were chosen as they are standardized, open, and widely used in the industry, when dealing with simulations and automation systems.

FMI defines a standardized interface for simulations in development of cyber-physical systems. Using FMI, an engineer can create a simulation module, containing all required equations and data for the simulation, called a Functional Mockup Unit (FMU). Such FMUs can be imported into other environments and be executed. The need for a unifying simulation standard comes from the state of the industry. The simulations often replace real testing to facilitate faster and cheaper development, however the simulation

landscape in the field of system and component design is very heterogeneous [BAS14] (as are the cyber-physical systems themselves [JOM16]). Different simulation tools are preferred for different engineering domains, and these tools have incompatible model representations. Some proprietary exchange formats exist, however they have limited functionality and are only applicable to some tool combinations [BAS14]. An open, more generalized approach, such as FMI would aim to alleviate these issues. First, the entire system should be modeled within one modeling language, this would greatly simplify the interaction between different simulated system components. This means that the modeling language must be suitable for a wide array of engineering domains. Modelica modeling language fulfills this [BAS14], and FMI provides the necessary features to transform Modelica models to FMUs [Mod20]. And second, the interface for model exchange should be standardized and tool-independent. This helps with the problem of heterogeneous simulation landscape, by allowing specialized tools for different domains, while maintaining interoperability. FMI is a good candidate for this kind of model exchange and cross-company collaboration [BAS14].

On the other side, OPC UA is often mentioned in the context of Industry 4.0 applications [GHIU17],[LKYO17]. For example, OPC UA is mentioned in the German Industry 4.0 Implementation Strategy from 2015 [BVZ15]. It offers client-server machine-to-machine communication, and Service Oriented Architecture (SOA). It also provides an extensible way to define information models related to a physical process. The need for such information models comes from the following: devices and subsystems of automation systems need to implement communication related functionalities, such as search, connect, send, and receive. While all of the subsystems could implement the same functionality, it would be more beneficial if these capabilities were implemented in a distinct system, which then provides interfaces to different subsystems [MGGU11]. Thus, the applications only need to handle interfaces, instead of individual functionalities. Obvious benefits are reuse, reduction of complexity, higher flexibility and maintainability [MGGU11]. OPC UA incorporates several functionalities which give "meaning" or semantics to the data, such as hierarchy, and inheritance. These capabilities help the information system fulfill its role regarding the existing data. Such machine-friendly semantic definitions can help software "reason" about the data at hand, to further help with the tasks of the information system. In other words, it allows clients to perform sophisticated tasks by interpreting the semantics of the data [MLD09]. OPC UA supports the following modeling concepts which are important for information modeling: hierarchy, variables, data types, functions, methods, references, classes, inheritance, and aggregation [MGGU11]; these functionalities help it fulfill the information system requirements.

OPC UA information models will be used, along with the model descriptions of FMUs to facilitate the mapping procedure, in the proof-of-concept implementation in this thesis.

Thesis Structure

The thesis is structured as follows: first, an overview of the current state-of-the-art is given in Chapter 2. The chapter also presents an introduction to the two frameworks for

simulations and automation systems which were chosen for the proof-of-concept: FMI and OPC UA, respectively. Chapter 3 gives additional insight into the problem, the possible workflow approaches, as well as some mapping (matching) techniques. Next, Chapter 4 presents the proof of concept implementation which was developed for the purposes of this thesis; the use cases which were used with the proof of concept are then described in Chapter 5. The findings of the thesis, and the proof of concept implementation are discussed in Chapter 6. And lastly, the thesis is summarized and concluded in Chapter 7, where some possible approaches for future work are also given.

CHAPTER 2

State of the Art

This chapter contains the related scientific work, as well as papers which provide additional information on topics that were mentioned in this thesis. Additionally, some basic information about FMI and OPC UA is presented, in order to make understanding the rest of the thesis easier.

2.1 Related Work

Industry 4.0

With the goals of higher operation efficiency, higher productivity, and higher levels of automation, the fourth industrial revolution aims to integrate different systems and build upon machine-to-machine communication. In his paper titled "Industry 4.0: A survey on technologies, applications and open research issues" from 2017 [Lu17], the author Yang Lu provides an overview of the scientific literature regarding the Industry 4.0. In total, 88 scientific papers were reviewed and summarized within this work. They are categorized in five categories which give the reader a good introduction depending on the particular direction they want to explore. The five categories are: Concept and perspectives of Industry 4.0, CPS-based Industry 4.0, Interoperability of Industry 4.0, Key technologies of Industry 4.0, and Applications of Industry 4.0. While the paper does not go into detail for each of the reviewed papers, it presents a good overview of the general topic of Industry 4.0, research directions, and a good introduction to Industry 4.0. Some of the major Industry 4.0 features presented in the paper, such as digitalization, automation, and automatic data exchange and communication, give a good motivation for this thesis.

Cyber-Physical Systems

Information about digital twins, their purpose and functionalities can be found in [SSK+20].

With regards to the cyber-physical systems, more information about the current and future aspects of CPS, based on an automotive system example can be found in [WMO+16]. The authors argue that the cyber-physical production systems, which are integral to the future factory environment (smart factories) should be scalable and modular. The advantages of these approaches include greater ease of integrating, adapting, and replacing individual production units, as required, based on the unpredictable market demands, or in order to maintain operability in the case of disruptions and failures. The authors pinpoint the concurrent combination of the physical world, and their digital counterparts as crucial in achieving these goals. Thus, the importance of simulation tools increases, as they support the engineering, re-engineering, and decision-making processes, as well as enable evaluation of external and internal changes, and the impacts they cause on the simulated system. Finally, in order to address the challenges of developing systems which can model and simulate components of future factories, the authors present a framework for modeling and simulation of CPS-based factories.
This paper provides a good motivation for co-simulation and digital twins, based on a real industrial example. This has inspired the support for automatic time progression clients, in the proof of concept implementation of this thesis (Chapter 4, in particular Subsection 4.2.2).

Functional Mock-up Interface

More information about FMI can be found in [BOA+11], or directly in the standard [Mod20].

Additional information on using FMI together with the concept of a digital twin can be found in [NFCM19]. The paper presents a modular simulation approach with the FMI for use in creating a digital twin, which is an integrated simulation of the system, or some component(s), which mirrors the corresponding real life component/system (i.e. the twin), using internal models of the real component/system, as well as available sensor data. This paper uses a black box approach for the individual modules within the main simulation models, which simulate different behaviors of the system. These individual models are activated only when needed, and work in addition to the main simulation model. This achieves additional flexibility of the simulation.
This paper presents a good motivation for the use of FMU modules. In the paper, they are added as black-box models to the main simulation model, in order to simulate different system behaviors (for example, energy consumption). These modules are then activated only when needed, making the simulation flexible. FMI allows the FMUs to be used by different simulation environments, granting them independence from a single simulation tool, and enabling relatively easy re-use. This paper has provided good motivation for the use of FMI and FMUs. The authors describe a laboratory example, where they use MATLAB Simulink[1] environment to construct the FMU modules (where this thesis, in contrast uses OpenModelica). Their example uses OPC UA to connect with an OPC

[1] `https://www.mathworks.com/products.html` ; accessed October 2021

UA server hosted on a PLC. The OPC UA connection is achieved through a MATLAB toolbox (in contrast to open62541 used in this thesis, Subsection 4.1.1).

OPC Unified Architecture

[LM06] presents a good overview of OPC UA, alternatively, one can find more information in the standard [OPC20] or in a book by Mahnke, Leitner, and Damm [MLD09].

More on information modeling with OPC UA in particular can be found in [GHIU17]. The authors consider the advantages and disadvantages of OPC UA information models. The paper presents an overview of different ways and projects where OPC UA information models were used, and how they impacted the corresponding system. One of the key advantages identified by the authors is the possibility for stronger decoupling of the client and the server in applications, while simultaneously making applications more flexible. In particular, the following use cases for OPC UA are considered by the authors: modular automation, co-simulation, and integration of field level devices with high data throughput. The authors also identify some of the disadvantages of OPC UA, such as limitations with regards to object orientation, server aggregation, and information model revisioning. Finally, the authors concluded that OPC UA, and its information models are suitable for new applications in the area of digitalization in process industries, by allowing the construction of flexible and smart applications.
This paper provides good motivation for the use of information models. Modularity and extensibility of OPC UA models are some of the advantages of OPC UA identified by the authors of the paper, and the proof of concept implementation of this thesis benefits from those advantages. In particular, the simulation control and simulation variables are kept in separate namespaces as they are parts of different information models (Subsection 5.1.2 and Subsection 5.2.2). Additionally, the existing OPC UA server in the proof of concept is relatively easily extended by the simulation variables, using the nodeset compiler which generates the entire simulation variable tree for the server automatically (Subsection 4.2.3).

Closer to the topic of this thesis, the possibility of integrating a dynamic simulator and advanced process control using the OPC UA standard has been considered in [LSS19]. The large amount of data generated by the manufacturing industries is difficult to exchange among the various manufacturing systems, applications, and components, due to the variety of the underlying systems. A middleware such as OPC UA could be used to exploit the existing data, and achieve higher efficiency by bridging the communication gap between the systems. OPC UA presents a scalable, platform independent, user friendly, and secure approach. The paper adopts a simplified version of the Tennessee Eastman (a chemical process simulation problem) - which is a well-known industry problem, as a use-case on which the simulation and control modeling is performed. The authors have also identified some of the difficulties when using OPC UA, such as the set of services in an OPC UA server which is used by the client is defined in the OPC UA standard and is not changeable by the user, and the security tiers are difficult if not impossible to control, after the application platform is chosen. The authors also point out that performance

measurement of the OPC UA connection is not present, and that the level of reliability and the quality of connection are hard to understand.
The paper successfully implements OPC UA communication between the controller and a simulator for the simplified Tennessee Eastman problem, however, it also identifies some difficulties when dealing with OPC UA. While the security and performance concerns about OPC UA do not influence the proof of concept implementation of this thesis, it is an interesting point to consider, especially regarding the access level criterion (Subsection 3.2.2), and larger use cases (mentioned in Chapter 6).

Interaction of OPC UA with another simulation environment than in this thesis, namely with VEROSIM[2], is handled in [RR20]. The paper concerns itself with the concept of automatic integration of simulation systems with OPC UA. A data mapping between the simulation meta data model, and the OPC UA information model is defined in both directions, and the communication is ensured with message passing between the OPC UA server and the simulation software. The simulation software chosen is VEROSIM - an extensible simulation framework. The mapping is used to automatically generate OPC UA address spaces from the simulation models, and the inverse mapping (simulation model to OPC UA information model) is used to generate simulation model structure from queries to existing OPC UA servers. A large number of concurrent simulated objects is possible, with individual and independent OPC UA servers and clients. Thus the approach of this paper is especially useful for the distributed simulation. Finally, the approach is tested in an automotive production line simulation, where the results show that the implementation can handle significant network load.
The most important difference of this paper to the approach of this master thesis is the fact that this paper presents a distributed solution with individual OPC UA servers, implemented for the VEROSIM simulation framework. Virtual reality and simulation system VEROSIM (proprietary product of VEROSIM Solutions) is a software solution for reality systems and geographic information systems. It was originally developed for the simulation of robot work cells for industrial applications [YJR+10], but it can now handle a wide variety of applications, such as simulations of agricultural machinery and the simulations of the international space station. VEROSIM puts a focus on 3D simulations and virtual reality. The core part of the VEROSIM is small and the functionality is added by plugins, this ensures a flexible, modular, and extensible design. An example for using VEROSIM for an extraterrestrial legged robot is given in [YJR+10].

More information about possible improvements to OPC UA communication can be found in [SFT+19]. This paper presents OPC UA extension where the information necessary for OPC UA server application to access the underlying system is specified in information model (called CommunicationInterfaceSpecification). The authors achieve this by defining communication interfaces for all nodes of the information model that represent readable or writable data, with the goal to reduce the implementation effort of OPC UA servers. The paper also presents different modeling approaches for communication interfaces and compares them. Three approaches for modeling the communication interfaces

[2] https://www.verosim-solutions.com/ ; accessed October 2021

are presented: as objects, as variables, and as data types. The functionality of the communication interfaces, as well as the hierarchy (CommunicationInterfaces folder and the Server object) are examined. If communication interfaces are implemented as objects, there is a useful feature - event notifiers, which allows OPC UA clients to subscribe to object nodes specified as EventNotifiers. Triggering conditions can be specified and they may then be triggered when a communication interface is added, removed, or changed. When modeling the interfaces as a variable, the communication interface information is defined by the variable properties, and the value of the variable itself can be used for additional information, for example the state of the underlying system. As an alternative to EventNotifier attribute, the events may be triggered by CommunicationInterfaces folder (from the hierarchy), so on changes to the folder, OPC UA clients will be informed about the changes to the communication interfaces. The DataType approach can only be applied in combination with a data variable or property, so it cannot be used as a standalone solution. The following criteria are defined: flexibility in integration and extensibility (objects cannot be instantiated under variables, this is a limitation of ObjectType approach), access control (here ObjectType approach has an advantage as individual variables access can be specified), adding/removing communication interfaces (variables and objects cannot be passed to OPC UA server as input arguments, DataType approach is favored), updating communication interface details (EventNotifiers vs subscribing to folder updates), support by OPC UA stacks and clients (not every OPC UA client may support structured DataTypes), support by SDKs. The recommended modeling approach is a combination of VariableType and DataType approaches. The paper also presents a proof of concept.
This paper presents good insight into OPC UA communication extensions. Using the presented Communication Interface Specification techniques to extend the OPC UA information model with the information required to communicate with the underlying system (in the case of this thesis, the simulation) would present some advantages for the proof of concept implementation of this thesis (for example: better organization and clearer communication requirements). However this would surpass the boundaries of this thesis; instead it could be an improvement avenue for future work.

2.2 Functional Mock-up Interface (FMI)

The version of FMI used in this thesis is FMI 2.0.2 [Mod20]. Both the FMI 2.0.1 and FMI 2.0.2 are maintenance/bugfix releases, and provide no new functional changes. The FMI 2.0, however, was a major expansion of FMI 1.0, as it merges the two previous standards, namely FMI 1.0 Model Exchange, and Co-Simulations, and it additionally incorporates many improvements.

2.2.1 Overview

In FMI, the Functional Mockup Unit (FMU) contains data that is relevant for a particular simulation (module). One of the included files is the model description XML file. This

file defines the variables, units, and other information that is relevant, and serves as the interface definition for the simulation model. Apart from the model description file, the FMU also contains the data required to simulate some instance/module of the system, according to an interface defined by the FMI. A simulation environment can utilize multiple FMUs or other models, and individual FMUs can be instantiated multiple times.

FMI for Model Exchange and FMI for Co-Simulation

There are two main use cases for FMI. First, FMI for Model Exchange which defines an interface to a model of a dynamic system which is described by differential, algebraic, and discrete-time equations. These models provide interfaces to evaluate these equations, which is a task for the simulation environment to perform. Such interface allows descriptions of large models. The FMU in this case contains the model, or communication to a tool that provides the model, while the environment provides the simulation engine. On the other hand, FMI for Co-Simulation is designed for both the coupling of simulation tools, and coupling with subsystem models. These models are exported by the simulation tools so that they include runnable code. In this case, the FMU includes the model and the simulation engine, or a communication to a tool that provides the model and the simulation engine; the environment provides the master algorithm to run coupled FMU co-simulation slaves together. The proof of concept implementation of this thesis (Chapter 4) utilizes FMI for Co-Simulation. In the case of simulation tool coupling, the simulation is performed independently for all subsystems, which limits the data exchange between the subsystems to discrete communication points. The visualization and post-processing of simulation data is then the responsibility of the simulation tool handling that subsystem.

Despite the different use cases, both the FMI for Model Exchange and FMI for Co-Simulation utilize many of the same parts of the FMI standard. In particular, the FMI Application Programming Interface is shared. All required equations, or other computation tasks are evaluated using standardized C functions. The FMI Description Schema (XML) is also shared. This schema defines how the XML file generated by a modeling environment should look (both in its structure and its content). All variables of the FMU are defined within this XML file in a standardized way, along with their attributes, such as name, unit, default initial value, minimum and maximum values, and so on. This is considered an advantage, as defining the variables and the relevant information directly in the C code would present an overhead for the embedded systems, and for large models. This also enables tools using different programming languages to easily utilize the variable information.

Properties and Guiding Ideas of the FMI

The standard also defines some guiding ideas for FMI. First is expressivity, meaning that the FMI provides necessary features for models from different vendors (including Modelica itself) to be transformed to an FMU.
Stability is naturally important for a standard which should be used by many simulation

tools worldwide, it is therefore a high priority, along with backwards compatibility.
Processor independence is also one of the desirable properties. This means that the FMU can be distributed without previous knowledge of the target processor. Processor independence increases the usability of an FMU, and is achieved partly by using C language for FMU source, which is highly portable.
The FMU should also have simulator independence. In other words, it should be possible to compile, link, and distribute the FMU, without knowing the target simulator. This presents a big advantage, as the FMU is not restricted at compile time, and there is no need for different versions of the same FMU targeted for different simulators.
The FMI has small runtime overhead, meaning that the communication between the FMU and the target simulator (through FMI) does not incur significant runtime overhead.
The FMI also has small footprint, in a sense that the compiled FMUs are small. This presents an advantage for embedded systems. Other relevant (required) information is stored in the model description file, which is also a part of the FMU, but not a part of the executable.
As was mentioned before, the FMI provides support for many FMUs, as well as for nested FMUs. This includes many different FMUs, or multiple instances of a single FMU. The inputs and outputs of the FMUs can be connected with direct feedthrough.
While an FMU normally computes a cache for later re-use, in order to simplify the usage and reduce the possibility of an error in the simulator, the caching mechanism is hidden. This aids the simplicity of FMI, and additionally it allows freedom of implementation of the caching policies for FMUs.
In order to minimize execution times, FMI provides support for numerical solvers, mainly through vectors of states, derivatives, and zero-crossing functions.
FMI consists of a few functions and avoids redundant functions which could be defined in terms of other functions. This results in a compact and easy-to-use API.
There is a unified error handling, meaning that all FMI methods use a common set of methods to communicate errors.
Memory management is handled with the Allocator Must Free policy. This means that the memory and other resources allocated by the FMU are freed by the FMU, and similarly, all resources allocated by the simulator are freed by the simulator. This helps prevent memory leaks and runtime errors due to incompatible runtime environments.
Finally, FMI is encoded in C and not in C++. Apart from the above listed benefits, this also avoids problems with compiler and linker dependent behavior.

2.2.2 Data Types

The FMI defines the five following types: `Real`, `Integer`, `Boolean`, `String`, and `Enumeration`. When the variables are defined in the model description file, in the FMU, along with the data type, other parameters can also be specified, such as the unit, minimal and maximal value, the nominal value, and others. An example of a variable definition in the model description file might look as follows:

```
<ScalarVariable
        name="mass.m"
        valueReference="19"
        description="Mass of the sliding mass"
        variability="fixed"
        causality="parameter"
        >
        <Real start="2.0" min="0.0" unit="kg"/>
</ScalarVariable>
```

Type definitions can also be given in the `TypeDefinitions` element, where one or more `SimpleType` elements can be defined. The variables are then given type via the `declaredType` attribute, which references one of the `SimpleTypes`. Each `SimpleType` has a `name` which must be unique among all `SimpleType` elements in the `TypeDefinitions` list. Additionally, `SimpleType` may contain a `description` attribute.[3] Additionally, a type must be specified, by defining one of the following elements: `Real`, `Integer`, `Boolean`, `String`, `Enumeration`. This type element (based on which one it is) then has additional attributes.
For example, a `Real` element may have a `quantity` (non-standardized physical quantity name, for example "Energy"), `unit`, `displayUnit`, `relativeQuantity` (these three are described in the Subsection 2.2.5), `min`, `max`, `nominal` (starting value), `unbounded` (in order to increase numerical stability of the simulation).
An `Integer` may only have `quantity`, `min`, and `max` attributes. While `Enumeration` may have a `quantity`, but it also has (at least) one, or more `Item` elements, which contain a `name`, a (unique within the enumeration) `value`, and (optionally) `description` attributes. An `Enumeration` might also have `min` and `max` attributes.

2.2.3 Variables

The variables within the model description file are defined within the `ModelVariables` element. This element defines one or more `ScalarVariable` elements. The scalar variables are uniquely indexed starting from 1 (these indices are not related value references which do not have to be unique, see Subsection 2.2.4). A scalar variable is a variable of a primitive type (for example `Real` or `Integer`, for variable types see Subsection 2.2.2). Only scalar variables are supported, which means that arrays or records have to be mapped to scalars (for more information about the variable structure, see Subsection 2.2.6).

[3] In order to preserve compatibility with other environments, such as Modelica, `name` of a `SimpleType` must also be different from all `name` attributes of `ScalarVariables`. This restriction is in place in order to prevent issues with FMU importing, as these environments often disallow instance names which coincide with type names.

Attributes

Apart from the data type definition, a scalar variable also has attributes. There are two obligatory attributes: the `name` and the `valueReference`. The name is, similarly to the index, a unique identifier of a variable. The value reference is used to identify the variable value within the model interface, but is not necessarily unique (see Subsection 2.2.4). Additional attributes of scalar variables are optional.

`description` is a freely chosen string describing the meaning of the variable.

`causality` defines several possible options: first `parameter` means that the variable is constant during the simulation, provided by the environment, and cannot be used in connections; `calculatedParameter` means that the variable is constant during the simulation, and computed during initialization or when tunable parameters are changed; `input` variables can be provided from another model or a slave; `output` variables can be used by other models or slaves; `local` variables are either calculated from other variables, or they represent a continuous-time state; and lastly `independent` variable is usually "time", as all other variables are functions of this independent variable (only one variable within an FMU can be defined as independent). The default causality is `local`.

`variability` attribute defines the time dependency of the variable (it defines the time instants when a variable can change its value). It allows the following options: `constant` means that the variable never changes; `fixed` means that the variable does not change after initialization; `tunable` means that the variable is constant between external events (in FMI for Model Exchange) and between communication points (in FMI for Co-Simulation); `discrete` means that the variable is constant between external and internal events (in FMI for Model Exchange), or the variable is from a real sampled data system where its value is only changed at communication points (in FMI for Co-Simulation); `continuous` (only possible for variables of type `Real`) means that the variable either has no restrictions (in FMI for Model Exchange) or that the variable is from a differential (in FMI for Co-Simulation).

The attribute `initial` defines how a variable is initialized, and allows the following options: `exact` means that it is initialized with the `start` value of the data type; `approx` means that the variable is an iteration variable of an algebraic loop, with the start at the `start` value; and `calculated` means that the starting value is calculated from other variables during initialization.

The attribute `canHandleMultipleSetPerTimeInstant` is only available for FMI for Model Exchange, and it means that the input variable with this attribute is or is not allowed to appear in a real algebraic loop requiring multiple set calls.

2.2.4 Value References

Each of the variables inside the model description file has to have a specified value reference, which is defined as an attribute `valueReference` in the element `ScalarVariable`. However, in contrast to one might expect, these value references are not necessarily

unique. The first possibility are the so called "alias" variables. These are the variables of the same data type, which have the same value reference. Other parameters of the variables, such as the minimal and maximal values may be different (but the [engineering] unit must also be the same). The values of these aliased variables must be identical. This means that the variables must adhere to the strictest value limitations that are present. For example, if two variables are aliased, and one has a maximum value of 5, and the other the maximum value of 10, then the simulation should trigger an error if the value of either of the variables (the same value) surpasses the lower maximum: 5. The second possibility for sharing value references is for the variables of different types. These variables can freely share value reference without any issue, as the interaction with the simulation (reading and writing variables) clearly specifies the type in the function call, which then ensures that the correct variable is accessed.

2.2.5 Units

The model description file contains a `UnitDefinition` section where units can be defined. Note that if no units are defined, the `UnitDefinition` element must not be present. The support for units is important for technical systems for ease of use and in order to reduce the chance of errors. As different systems use units differently, there is no unified way to handle units which is satisfactory for all applications, and the unit handling is thus a difficult topic. A single `Unit` is defined by its `name` attribute, which must be unique among all units (i.e., a `name` of a `Unit` uniquely identifies it). If a variable is defined to have some unit, the value of this variable when interacting with the FMU (i.e., setting and getting the value) will always be with respect to the unit associated with the variable.

Additionally, it is possible to define how a `Unit` can be converted to a `BaseUnit`. The `BaseUnit` definition has exponents of the 7 SI base units (kg, m, s, A, K, mol, cd) as well as an exponent of the SI derived unit rad (this helps with quantities that depend on angles, which often occur in technical systems). `BaseUnit` definition also includes the `factor` and `offset` attributes, which together form a linear conversion formula: `BaseUnit_value = factor * Unit_value + offset`, which can be used to convert the given `Unit` to a base unit.[4] For example, a Joule (J) is $\frac{kg*m^2}{s^2}$ in SI units. This would mean that `Unit.name` would be `J`, and the `exponents` of `Unit.BaseUnit` would be `kg=1, m=2, s=-2`, the `factor` would be `1.0` and the `offset` would be `0.0`. The `BaseUnit` element can be used for the signal connection check. If two signals are connected together (for example, an output of one FMU is connected to an input of another FMU), and both of them have `BaseUnit` elements defined, then they must have identical `exponents` (thus, wrong connections can easily be detected). If `factor` and `offset` attributes are also identical, the values must be the same, otherwise, the system might trigger an error, or perform a conversion, if supported. Caution must be exercised with variables that have the same `BaseUnit` but

[4] If `offset` should be used or not is determined by the `relativeQuantity`, which is an attribute of the `TypeDefinition` of a variable

belong to different quantities (for example energy and torque). In such cases, quantity definitions need to be taken into account. Units might also be deduced based on signal connections for variables that are missing unit definitions.

Optionally, it is also possible to define a set (one or more) of display units. A `DisplayUnit` is defined by `name`, `factor`, and `offset` attributes. The `name` attribute must be unique within other `name` attributes in `DisplayUnit` definitions of the same unit. The conversion then works similarly to the `BaseUnit` conversion: `DisplayUnit_value = factor * Unit_value + offset`.

2.2.6 Variable Structure

Being able to form arrays and other structures is often an important part of models, as it can make the understanding of the model easier for humans, but also enable additional matching techniques for programs, such as those handled in Subsection 3.2.2. In the model description file in the FMU, there is a parameter called `variableNamingConvention` which can be given as "flat" or "structured". Using flat variable naming convention, the variables simply represent a list of strings. The names of the variables using flat variable naming convention can be any Unicode characters apart from: carriage return, line feed, and tab. On the other hand, if structured variable naming convention is used, the variables follow a certain hierarchy. Namely, the hierarchy is established by using a dot (".") as a hierarchy separator, using square brackets for array elements, as well as designating derivatives of other variables with "der()" (higher derivatives are given as "der(name, N)", for N-th derivative. Other than these hierarchical specifiers the names can consist of underlines, letters and digits, or alternatively, any characters enclosed by single apostrophes. When defining an array, all array elements should be given as a consecutive sequence of scalar variables. In a case of a multi-dimensional array, they are ordered row-major. The standard specifies the following example:

The vector "centerOfMass" in body "arm1" is mapped to the following ScalarVariables (it is not defined if the arrays start with 0 or 1):

```
robot.arm1.centerOfMass[1]
robot.arm1.centerOfMass[2]
robot.arm1.centerOfMass[3]
```

2.3 Open Platform Communications Unified Architecture (OPC UA)

This thesis uses the 1.04 version of the OPC UA standard [OPC20]. The OPC UA standard is split into several parts, each focusing on a certain aspect of the standard. The parts from 1 to 7 (Overview and Concepts, Security model, Address Space Model, Services, Information Model, Service Mappings, Profiles, in that order), including part 14 (PubSub) focus on the core capabilities of OPC UA, such as the structure of the Address

Space, and the available Services. Part 14 focuses specifically on the publish/subscribe pattern (whereas Client/Server pattern is defined by Services in part 4). Parts from 8 to 11 (Data Access, Alarms and Conditions, Programs, Historical Access, in that order) then apply these capabilities to specific access types (which were previously addressed in separate OPC COM specifications). Parts 12 and 13 (Discovery and Aggregates, respectively), are considered utility specification.

2.3.1 Overview

The intention of OPC UA is to be applicable to all industrial domains, from industrial sensors and actuators to enterprise resource planning and manufacturing execution. In particular the standard mentions Industrial Internet of Things, Machine To Machine communication, Industry 4.0, and China 2025 (a comparison of Industry 4.0 and China 2025 can be found in [Li18]). The focus is on information exchange together with command and control for industrial processes. This information exchange is facilitated by defining a common infrastructure model which specifies the following: the information model which represents the structure, behavior and semantics, the message model which facilitates interaction between applications, the communication model in order to transfer data between end points, and the conformance model to guarantee system interoperability.

The main representation of the OPC UA information model is the nodeset file, which is an XML file containing the information about the nodes, variables, and parameters, which constitute an information model. The nodeset files serves as an interface for the automation system information model.

OPC UA aims to be platform independent standard, to enable communication between different system and device types. On a technical level, OPC UA standard defines several communication protocols: OPC UA TCP, HTTPS, and WebSockets. The communication is facilitated by one of the following communication types: either by sending request and response messages between clients and servers (defined in standard part 4), or through `NetworkMessages` between publishers and subscribers (defined in standard part 14). The Client/Server model defines a set of services that a server may provide, whereby individual servers specify the service sets that they support, to clients. Servers define object models that clients can dynamically discover. The data that the servers provide is not limited to the current ("live") data, as the servers can also provide historical data, as well as alarms and events. The portability and the flexibility of OPC UA is achieved by mapping OPC UA to a variety of communication protocols. The data can also be encoded in various ways, in order to ensure portability and efficiency. The advantage of offering multiple possibilities is that end users can make decisions about performance and compatibility trade-offs at deployment, as opposed to having the encodings and protocols pre-determined by the vendor at product creation.

The service and address space model is consistent and is provided by the standard. A single server can then integrate the data, history, alarms and events into its `AddressSpace`, in order to provide access to them, using a set of `Services`. The services also include an

integrated security model. The communication is robust and secure, assuring the identity of OPC UA applications. The robustness is also guaranteed by mechanisms which ensure that clients can quickly detect, and also quickly recover from communication failures associated with data transfers (avoiding the long timeouts of underlying protocols). The servers can also provide clients with type definitions for objects from the address space, this means that the information models can be used to describe the contents of the address space. The data itself can be presented in different formats, for example binary structures (UA Binary), or XML or JSON documents. The format of the data itself can be defined by OPC, as well as other standard organizations or vendors. This guarantees the extensibility and flexibility of the standard. The data is accompanied with metadata, which describes the data format. This metadata can also be accessed by clients, by requesting it from the server. This facilitates flexibility, as a client with no previous knowledge of data formats can determine the format of the needed data at runtime, which enables successful utilization of the data.

Many relationships between nodes of an information model are supported within OPC UA. This means that the server can present the data in a variety of hierarchies which can be specifically tailored to sets of clients that view the data. The OPC UA is designed in such a way to support a variety of servers, fitting many industrial niches (from the plant floor to enterprise). Supporting such a wide variety of use cases means that the servers can differ in size, performance, execution platforms and also functional capabilities. The OPC UA standard aims to isolate the core design and ideas of OPC UA from the underlying technology and networks. This ensures easy adoption and adaptation of OPC UA to new technologies without changing the core design. In order to ensure interoperability, OPC UA provides a set of capabilities, which the servers may implement, in addition, OPC UA defines specific subsets, called `Profiles` (standard part 7), which the servers can then support. This enables easy interaction with the clients, as the clients can detect server profiles, and then adjust their interaction with the server, based on those profiles.

Communication types

The clients and the servers are defined as interacting partners within the OPC UA systems architecture models. A system may consist of multiple clients and servers, where each server can simultaneously interact with one or more clients, and each client can simultaneously interact with one or more servers. A single application may have both server and client components, in order to allow interaction with other clients and servers. This is called server to server interaction (one server acts as a client of another server). Server to server interaction enables the servers to exchange information with each other on peer to peer basis, and it also enables server chaining in a layered architecture.
The server has an `AddressSpace`, which is modeled as a set of nodes which are accessible by the clients, using services. The address space contains the nodes, as well as the references between the nodes. The server is free to organize the nodes within the address space freely, using the references, which enables building hierarchies, mesh networks, or something in between. A `View` is a subset of the address space, which the

server can use to restrict the size of the address space for the clients. Views are often used as hierarchies which make the address space easier for the client to navigate.
The client interacts with the server through send and receive service requests and responses. The OPC UA communication stack converts client calls into messages, which are sent through the underlying communication system to the server. The OPC UA communication stack also receives response, as well as notification messages, from the underlying communication system, and delivers them to the client application.

The publisher/subscriber model helps OPC UA with expanding into new fields. The publisher/subscriber model is not restricted to a pre-defined message system (so both conectionless protocols such as UDP can be used, as well as established message protocols, such as ISO/IEC AMQP 1.0). This communication model can be useful to facilitate configurable peer to peer communication between controllers, for example, as the data exchange often needs to happen within a certain time frame. The publisher/subscriber model may also be used for asynchronous workflows, where an application may place its message in a message queue, from which it can be handled by one of several parallel applications, running on the other end. Sensors and actuators can also follow the publisher/subscriber model.
Using the publisher/subscriber model, the OPC UA applications do not directly exchange message requests and responses, instead the publishers send messages to a message oriented middleware, without any awareness of the possible subscribers. The subscribers express interest in specific types of data, and process such messages, without the knowledge of the publishers. This message oriented middleware is a hardware or software infrastructure which supports message exchange for distributed systems. The message exchange implementation is not restricted by the standard.

2.3.2 Address Space

The goal of the address space is to provide a standardized way in which the servers represent objects to clients, as defined by the OPC UA object model. In other words, the set of objects and related information, that the server hosts is referred to as the `AddressSpace` of the server. The objects are defined in terms of variables and methods, and the address space allows defining the relationships between objects. The nodes are assigned to node classes, which represent different elements of the object model. The elements of the object model are represented as nodes, within an address space (within an `AddressSpace`, `Nodes` are described by their `Attributes`, and are interconnected by `References`). Attributes are data elements which describe nodes, they are accessible by clients using read/write, query, or subscription/monitoring services. The references are used to relate the nodes to each other, and they are defined as instances of reference type nodes.

Variables are used to represent values. OPC UA defines two variable types: `Properties` and `DataVariables`. Properties are characteristics of objects, data variables, or other nodes. A property characterizes what a node represents. The properties can be defined directly by the server, in contrast to attributes, which are common to all nodes of a

node class. Properties are not allowed to have other properties defined for them. Data variables on the other hand represent content of an object. An example would be an object which contains a stream of bytes, while the creation time and owner of the file would be properties, the data stream (stream of bytes) itself would be a data variable. Data variables might contain other data variables.

Methods are specified by the standard as lightweight functions, whose scope is limited by the corresponding object, or an object type. The execution of a method works as follows: a method is invoked by a client (method call), the method is then executed by the server, upon completion of the method, the server returns the result to the client. The methods are stateless, however, they can influence the state of the corresponding object. Each method may have multiple input arguments, as well as multiple output results. The methods are discovered by the client by browsing the objects to which the methods are attached.

Node IDs

The OPC UA nodes (and thus variables) are distinguished by the node ids (which contain namespaces). According to the standard, a node id is: "An identifier for a node in the address space of an OPC UA Server". A node id consists of the following information: an unsigned 16 bit integer representing the namespace (whereby the index 0 is used for OPC UA defined node ids); and a node identifier, along with its type. The node identifier can have the following types: `UInteger`, `String`, `Guid` (a 16 byte value that can be used as a globally unique identifier), or a `ByteString` (a sequence of bytes). A particular combination of the namespace and the node identifier is unique among nodes.

2.3.3 Data Types

All OPC UA types stem from the `BaseDataType` type. The data types are defined in OPC 10000-5. Some subtypes of `BaseDataType` are: `Boolean`, `ByteString`, `DateTime`, `Enumeration`, `NodeId`, `Number`, `String`, and so on. Some of these subtypes are abstract, and have further subtypes, for example `Number` has following subtypes: `Integer`, `UInteger`, `Double`, `Float`, and `Decimal`. And some of those have further subtypes, for example `Integer` which defines integers of certain bit sizes. Worthy of note is that the types are defined in a fine-grained manner, allowing precise type definitions for individual variables.

2.3.4 Structure

There are many reference types within OPC UA, allowing for relatively complex structures. The main structure consists of a Root folder, containing Objects folder, where objects instances are defined. The most common reference types here are `HasComponent` and `HasProperty`. The objects themselves may have other objects as components, and variables as components or properties, the difference being that properties cannot have further children of their own. Arrays are distinguished from scalar variables by using

the value rank attribute, which can be used to define the number of dimensions that the array has. Another attribute called array dimensions can then be used to define the exact length in each array dimension. Another common type is a method type, representing an executable method, which can have arguments and return values.

2.3.5 Security

The security in OPC UA (standard part 2) concerns itself with the integrity and confidentiality of communications, as well as the authentication of clients and servers, and authentication of users. Security profiles are defined in standard part 7. In order to ensure flexibility, the OPC UA standard does not specify the circumstances under which particular security mechanisms are required, instead, the security model of OPC UA defines which security measures can be selected and configured. The decisions about which security mechanisms to use are made by those who design and implement the system (or by other standards that they are following).

Application layer security is granted by a secure communication channel which ensures the integrity of messages. The user authentication occurs only once, when the session is established, the secure communication channel then guarantees the integrity of all messages exchanged within the session.

Sessions are logical connections between clients and servers, and are independent of the underlying communication protocols. They give the need for a stateful communication model (for server/client). The information related to the session are subscriptions, user credentials, as well as points of continuation for operations that span multiple requests. The servers can also limit the number of concurrent sessions, for example, based on the resource availability. As the sessions are independent of the underlying communication protocols, the termination or a failure within the underlying communication protocol does not cause session termination. Sessions are terminated solely based on the requests of either the client or the server, or based on the inactivity of the client (for a pre-determined time period, negotiated at session establishment).

CHAPTER 3

Proposed Interfacing Approach

This chapter gives a more detailed look at various engineering workflows. The problem can take several forms, depending on the design of the systems. The design and the chosen approach influences the matching, which needs to take place in order to enable seamless communication between the automation and simulation systems.

3.1 The Engineering Workflow

The key problem that is discussed throughout this thesis is as follows: there is a simulation environment on one side and an automation system on the other. The two systems do not have inherent interoperability, however they need to interact, as the simulation represents or replaces a real part of the system. The overall system thus consists of three components. First, the simulation environment models either a component or a set of components. The simulation either has the purpose of replacing the component temporarily in a real system (hardware in the loop), testing the behavior of the rest of the system before a real component is available, or simply doing tests on the simulated component itself, to see how it would behave. The second component is the automation system, it contains the means to control and optimize the process, and it interacts with the real hardware. The third component, and the focus of this thesis is the mapping unit. It is the unit that connects the other two sub-systems together, and the one which serves as a connector by enabling communication between them.

The problem itself can be tackled from multiple directions. This depends on the available system parts, the order in which they are developed, as well as the influence of the mapping unit designer over the design of the rest of the components.

The first possibility has the automation system designed first. This possibility can be seen in Figure 3.1. The full lines represent communication, the dashed lines represent design flow.

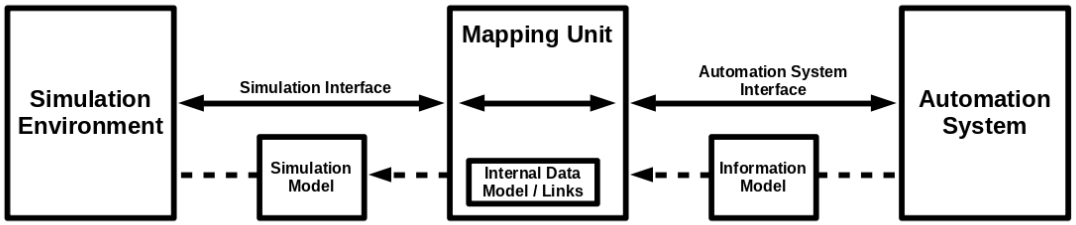

Figure 3.1: Engineering Workflow: Automation System First

Based on the automation system, one would create a mapping unit, for the future simulation. The mapping unit designer would thus create requirements for the simulation, such as which variables need to be accessible, their types and so on. The problem with this approach is that the requirements for the simulation system might be unfeasible to fulfill for the simulation. Not all simulation systems allow users to manually determine all variables and their types - the variables are usually generated by the simulation compiling software automatically.

The second possibility is to have a simulation model first, and then create a mapping unit, which would then set requirements for the automation system. This can be seen in Figure 3.2.

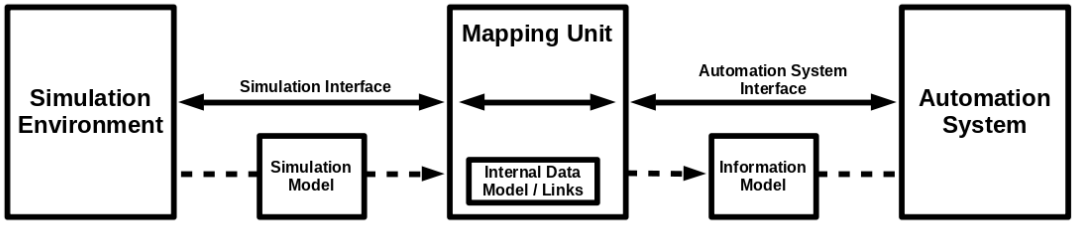

Figure 3.2: Engineering Workflow: Simulation System First

While the automation system would be easier to design according to these instructions (compared to the simulation system from the first possibility), the designers of the mapping unit would need to have good knowledge of what the automation system will look like in the future, and this is usually not directly possible. So, if the automation system would later need some additional variable, as the simulation and the mapping unit are already designed, providing it would present an issue. The advantage of both of these possibilities is that the matching is much easier to perform, as the mapping unit (designer) is the one who creates the requirements for one of the systems. This means

that the mapping unit knows exactly which variables will appear, and exactly how to distinguish them, making matching significantly easier. For example, the requirements may define exact variable names for each of the new system's variables, which would then make matching based on variable names trivial. On the other hand, both have the disadvantage of a more difficult (or unfeasible) system design, for the system which has to adhere to these requirements. But the real issue with both of these approaches is that it is usually the case that the automation system and the simulation models are already present (brownfield approach, see below in this section), thus we cannot design one of them based on the other one, and we are left with the third possibility.

The third possibility would be to have the automation system and the simulation system designed first and in parallel, and then design the mapping unit to connect the two. This approach can be seen in Figure 3.3.

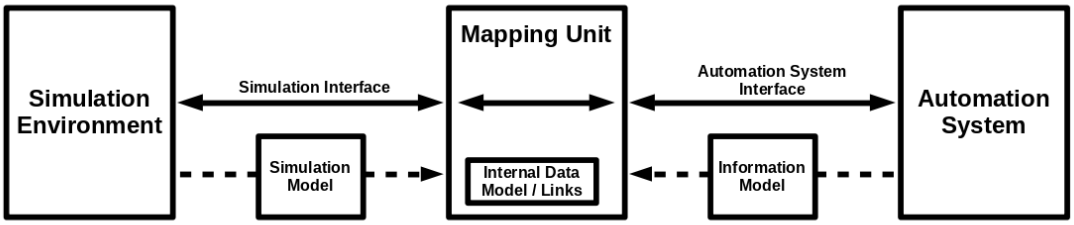

Figure 3.3: Engineering Workflow: Mapping Unit Last

This is the approach which is taken in this thesis, and assumed in the proof of concept implementation (Chapter 4). It often happens that the automation system, and the simulation are developed separately (also apart from each other in time). The systems should then be used together, however they are not developed with interoperability in mind. This means that this approach is the only viable option. The difficulty here comes from the mapping unit design. This approach then needs good matching techniques (some are handled in Section 3.2), in order to match the variables of which the mapping unit has no previous knowledge of. The mapping unit also did not take part in designing the requirements for either of the systems (or their interfaces). However, one advantage that the mapping unit has here, is that the human intervention is relatively easy to introduce, at least compared with the interfaces of the other two systems, which may not allow modification after the design is done. This means, that if the matching unit does not manage to match all variables automatically, the user might give the matches for the remaining variables, based on their knowledge of both the automation and the simulation system. This approach may be required (i.e., the only possibility), as it is often the case that the automation system and the interface of the simulation system already exist (as legacy systems), so the only factor missing is the mapping unit to connect them. An issue that might arise while using the method is that as the interfaces of the systems are

completely independent, it might be the case that the variables required by one of the system are not available in the other. The matching unit would then have no possibility of matching such variables.

It is also important to mention the greenfield/brownfield differentiation. Greenfield refers to the approach when the design starts anew, from the beginning; brownfield refers to the approach when some parts of the system are already present, and the main work is on expanding the already existing system. Starting everything from the beginning would provide many advantages, in particular, it would combine the advantages of both of the partial approaches (simulation system first, then mapping unit; and automation system first, then mapping unit). Namely, it would be possible to define exact requirements for both of the systems, in order to make matching trivial. The issue here is that we may not have much knowledge about the systems when just starting the design, and this might lead to combining the disadvantages of the two approaches, i.e. we may present unfeasible requirements for both of the systems. In order to avoid this issue, the entire design must happen simultaneously, and cooperatively, in order to ensure that the requirements can be fulfilled. Such design might be more complicated than "isolated" system design. However, generally, the greenfield approach is still considered more favorable.

On the other hand, if we have the brownfield approach (meaning that the system is already in place), we would first need to consider what is already developed. In a sense, the three cases which were described above are all brownfield cases, and they differ in what is already in place. What is common for the brownfield approaches is that the new (to-be-designed) part of the system needs to adapt to the existing part of the system. So, despite having the advantage of already having some sub-systems designed and finished, there is an additional challenge to adapt to the existing systems, which were possibly designed without the particular extension in mind. The brownfield approach is often what one has in practice, and especially the third engineering workflow from above where both the automation system and the simulation are already developed. The proof of concept will thus focus on this workflow.

3.2 Information Matching

This section concerns itself with the topic of matching. Matching in this sense refers to finding corresponding (correct) pairs of variables; in other words, one simulation variable, and one automation system variable which both represent the same data. Finding these matches automatically is not trivial, and this section will explore some of the possibilities to do so.

3.2.1 Approach

The decision to approach the problem from one of the sides (search for automation matches for simulation variables, or vice versa), can be taken, however it can partly limit the efficiency of the match search. The best possibility is to use the information from both

of the sides to find possible matches. However, it is probably beneficial to start searching for matches from the automation system side, and to consider the matching done, once all relevant automation system variables have their match. The reason for this is that in real simulation systems, there can thousands of simulation variables which are not directly relevant for the outside automation system (and thus the user), and are only used within the simulation internally. This issue was encountered while using OpenModelica[1], and is discussed in more detail in the Chapter 6. Many of these simulation variables do not have a match. The information which variables are to be ignored is important for the matching program, as it reduces the number of variables that need to be matched, making matching the rest of the variables easier. While it is beneficial to start looking for matches from the automation system side, the problem should sometimes also be looked at from the simulation system perspective, enabling different observations. For example, there might be a simulation variable which only has one possible automation system match, it is then highly likely that this is the match for this variable.

Another issue to consider when writing the matching program is the probability (not in a statistical sense) of a match. This issue is explored in more detail in its own subsection (Subsection 3.2.3).

Lastly, an option for human intervention should be given within the matching program. At the end, the program could make mistakes (for example with aggressive assumptions) or may not be able to match some variables. The program should then allow the user to manually specify which variables match to each other, as the user ultimately has more general knowledge of the systems, and what the modeling objective is. In order to make this easier, the program should provide a list of high probability matches to the user for the variables that were not matched by the program. This makes manual specification much easier, if the user only has a few variables to go through to determine the match, instead of hundreds, for example. A possibility for how this would work is given in the proof of concept implementation, and can be seen in the corresponding chapter (Chapter 5), Subsection 5.1.3. The user can then lean on their global knowledge of the models, presented information, and possibly variable descriptions, to determine the matches.

3.2.2 Possible Matching Criteria

This subsection will explore some of the possible criteria when determining variable matches. Using as many criteria in the matching program as the system design will allow is beneficial, as it increases the certainty of the match. As already mentioned in the previous subsection, the priority of the criteria and how much weight is given to each category when determining the probability level of a match can be an important aspect of the design. Ultimately, choosing the criteria and their number well will yield a program which is able to make many and correct matches; choosing the criteria poorly,

[1] `https://www.openmodelica.org/` ; accessed April 2021

and have them sparsely used in the system design will yield a program which struggles to make matches.

Data Types

Arguably the most important criterion is the data type. While the rest of the criteria may or may not be defined, most of the variables will have some form of data type associated with them. While having the matching data type might not indicate a certain match (there is probably a high number of variables with the same data type), having a data type mismatch indicates that the match has a very low likelihood. Note that a matching data type does not refer only to the exact same data types. For example, a 64bit floating point (double) variable in one of the systems might be the only possibility to which a 32bit floating point (float) variable in another system could match, as the first system maybe does not provide 32bit floating point type. In order to determine appropriate correspondence of the data types, a good knowledge of both of the systems is required. As stated before, adhering to some rule set during system design is also important. For example, one might use a 32bit signed integer for the number of processing steps in an industrial process, despite the fact that the number of steps is, for example between 2 and 7. Ignoring the fact that the system also provides smaller integers when 32bits are not required (as is the case in this example), and also unsigned integers, when the negative value is impossible, one might be tempted to use 32bit signed integer for all integer requirements. However, such decisions, would result in a bad design, making the matching process more difficult, as, in the example, such variables are first checked against 32bit signed integers in the other system, instead of 8bit unsigned integers. Choosing the correct data type at system design is thus crucial, however not all systems provide fine-grained and detailed data types.

Another criterion closely connected with data types is the variable range. This refers to the minimum and maximum values that a variable might take. This information, of course, makes matching easier, however it might also benefit systems where one of the system's data types are more coarse than the other system's. For example one system might provide 8bit, 16bit, and 32bit signed and unsigned integers, while the other system only provides signed 32bit integers. In order to find a match for the first system's unsigned 8bit variable, one might search through 32bit signed integers in the second system which have a minimum value of 0 and the maximum value of 255. The range information thus not only helps matching the variables with same range restrictions, it can also help match variables with different data types in systems which do not offer the same data type choices. Additionally ranges might help the user when manually determining variable matches. An example for this would be two temperature variables, one in Kelvin and one in Celsius, where the engineering units are not specified. The user knows that one of the variables is in Kelvin and another in Celsius from the more detailed description in the other system. If one of the variables has a minimum value of 0, and the other has a minimum value of -273.15, then it is easy to determine that the first variable is in Kelvin, and the second one in Celsius. This example shows how

one criterion might help cover for the shortcomings of another - in this case the user has used the range information to overcome the lack of engineering unit information in one of the systems.

Engineering units

Another important matching criteria are the engineering units. In simplest terms, having the same engineering unit makes the match more likely. Engineering units are a quick way to determine what a variable represents (or what it does not represent). If one of the variables has the engineering unit of kilogram, and another one has meters, it is certain that they are not a match, as one represents mass, and the second one length. While relatively straightforward, there are a couple of difficulties that might be encountered when using this criterion. The first issue is the possibility of different engineering units for the matching variables in two systems. For example the automation system might have the temperature in Celsius, while the simulation system internally uses Kelvin. It is however debatable if this would be a design failure, as the variables are not a direct match (their values would not match). In order to fix this, one would either need to fix the design, or to provide some kind of conversion functionality between the units. Additional difficulties might present themselves in this case, if for example, we are presenting mass of some object with a granularity of one gram, one variable might be an integer in grams, and another one might be real number in kilograms. This would further complicate the matching process, and should be avoided, if possible, at design. As these variables do not present a direct match, it is easier to design the systems correctly (possibly following some rule set), than to make complex correspondence and conversion rules; however, it might be the case that the systems are already given, and that the matching program designer has no influence over them, in which case correspondence and conversion rules might be necessary.

A second issue that occurs when using engineering units criteria is the engineering unit representation. Namely, two variables might have the same engineering unit, but with a different representation. For example, one system might represent degrees Celsius as "degC", and another as "°C"; or `"m/s2"` and `"m.s-2"`, both representations for $\frac{m}{s^2}$. One of the possibilities is to make sure that the engineering units match exactly at system design, this would also make the designs easier to read by humans. The second possibility (if for example the system design is already done), is to make a conversion table between the two units, tailored specifically to the problem at hand, for example in the form of a conversion table. Finally, a more general solution would be to convert all engineering units to a standardized representation of that unit, and use that for matching. The conversion tables for this case would need to be system specific, but they could then be reused for that system units regardless of the other system and its representations. This solution is favored for more complex systems, as the conversion tables can be easily reused. An example for such standardized conversion table is UNECE to OPC UA[2].

[2] `http://www.opcfoundation.org/UA/EngineeringUnits/UNECE/UNECE_to_OPCUA.csv` ; accessed April 2021

Access Levels

This criterion can be used to alleviate one of the problems mentioned before, namely the large number of simulation variables. In general, the simulation variables can be partitioned into three groups: the input variables, the output variables, and internal parameters. Depending on system design, it might be the case (and is desirable) that the variables that are relevant for the automation system (and thus the matching) are only in input and output variable groups, while all other variables - the internal parameters are not relevant to the automation system and do not need to be matched. Having such system design would simplify the search for matches by greatly reducing the number of variables that need to be matched. In order to additionally simplify the matching, the automation system might provide some of its variables at "read-only" access level. These variables then almost certainly correspond to the simulation's output variables, and do not correspond to the input variables. Thus, using access levels can greatly simplify the matching process, however, the steps need to be taken during system design, and changing the system design might not be possible at the point of matching. Additional category that might appear are read-write variables, which do not simplify the matching, due to their general nature, however they might indirectly simplify the matching in a sense that they probably do not correspond to the output variables, for example, as writing to simulation output variables would make no sense. It is also important to distinguish between different access levels within the automation system. What was referred to so far are automation system client access levels. The part of the automation system which interacts with the simulation (most likely the server), needs the opposite access rights of the client. For simulation's input variables, the server needs to be able to read the automation system variable, and write it to the simulation; and equivalently, the server would need to write to the automation system "read-only" variable, after reading the value from the simulation.

Structure

The variables within different systems are often structured and do not just present a flat list. This structural information could be used to facilitate the matching process. Instead of matching individual variables, the whole structures are matched, this means there are multiple variables which are matched at the same time, thus greatly increasing the confidence in a match, if one is found. The first, and the obvious issue here, is that the structures within different systems might not coincide. The simulation system and the automation system might have different ideas of how the modeled system should be structured, and as a result they may have different structures. This of course makes structural matching much more complicated and reduces the confidence one might had in matches. The second issue that appears here are repeating structures. Namely, one might have, for example a fan model, which has several variables under it, such as speed, voltage, and so on. This fan model could appear numerous times within the system model. Determining which simulation system fan corresponds to which automation system fan is a difficult task, that without additional information most likely requires

human intervention. However, presenting data structured in this way might make the job easier for the user who has to manually match the structures, as entire structures can be considered at a time, possibly matching a large number of variables at once.

Variable Names and Descriptions

Another criterion that highly depends on system design are the variable names. In a trivial example, if one has control over both of the designs, the matching variables are given the same names, and the matching program could be written in such a way to give the highest priority, or only consider name matching. However, apart from this trivial case, the name matching should be used sparingly. If the designs are not directly controlled, there are no guarantees that the variables will have the same, or even similar names. Where the names might help, is when a match for some variable is not found, and when human intervention is required. In this case, the user is provided with a list of high probability matches, the search for the correct match might be simplified by sorting the possible matches by name similarity. Additional attribute that is often present, and could be used in a similar way (being more helpful for the human) are variable descriptions. They might help the user with general understanding of the variables and make the manual matching easier, however unless strict design rules are used for them, they are not of much help for automatic matching.

3.2.3 Matching Probability

It is first important to state that the matching probability does not refer to a probability in a statistical sense. Instead, it represents different confidence levels to how likely a certain possible match is. For example, if one of the possible matches fulfills all of the matching criteria, and a second possible match fulfills only half, then we can say with reasonable confidence that the first match is more likely. However, there might be possible matches with different fulfilled matches where one is not a true subset of the other. In this case, determining which of the matches is more likely is not trivial, and solving the problem might require human intervention. Despite this, some criteria carry higher probability weights than others; for example, a real number variable on one side is very likely to match with a real number variable on the other side, and not with an integer variable, or some other type. In order to achieve easier matching, and to be more certain in the taken choices, both of the systems need to provide as much information as possible, and adhere to some design restrictions. For example, it might be clear for the user from the name of the variable "car_mass" that the value should be a real number, expressed in kilograms (this assumption depends on locality - another user might assume that the variable should be expressed in pounds), and that the value cannot be negative; however, if this information is not provided to the program, the matching process will probably be much more difficult. To this end, it is important that the variables and their properties are correctly and accurately labeled, on both the simulation and the automation system side. In addition, adhering to some standard or rule set when creating the systems would greatly improve the feasibility of matching.

In order to formalize the matching probability, one might develop a numerical value W which gives weights to all variable combinations: i and j, where i is the variable index from the simulation system, and j is the variable index from the automation system. For each of the combinations, one would need to go through all criteria which were chosen, and determine the value of that criterion for the particular pair ($c_q(i,j)$ for the criterion q). Each of these criteria would then be weighted according to their importance (with w_q). Such formula might look like this:

$$W_{i,j} = \sum_q w_q * c_q(i,j)$$

It would be possible to incorporate the criterion weight within the criterion value, however using them separately makes adjusting these weights clearer (and if desired for a more compact formula, they can be brought into one). Another reason why it is beneficial to keep the values separate, is that the criterion value c_q could be a simple boolean check for if the variables fulfill the criterion or not. For example, criterion t could test if the variables have the same type, in which case c_t would be 0 if variables i and j have different types, and 1 if the types match. This allows to separately set the weight and importance for criterion t, which in the case of the type should probably be high.

On the other hand, it is possible to also have the criterion on different levels, depending on the relation between the variables. For example, if we have two variables that are a match: a 16bit unsigned integer on the simulation side and a 32bit signed integer on the automation system side; the difference in types might come from different knowledge levels about the variables, but also the particular range might be incorporated in range information for the 32bit signed integer, where it could limit its values to be in the range of 0 to 16bit unsigned integer max. In any case, it is a higher probability that these two variables (32bit signed integer and 16bit unsigned integer, without other information) match, than that one of them matches to a boolean, or a string. Note that strictly taken, matching an integer to a string or a boolean is possible too, as one might represent integer as a string, or the value might just be a true or false, but the programmers used integer type for some reason. Obviously, such possibilities highly complicate the matching process. Depending on the engineering workflow (Section 3.1), the mapping unit designer might have the knowledge about variables, and if such different type matches can appear; or in cases of other engineering workflows, where the mapping unit is not designed last, set the requirements so that other system designers have to adhere to a certain typing convention.

With the expanded knowledge, the mapping unit designer can thus know which criteria can never be broken, in a certain match. For example, if all engineering units in both systems are always defined using SI[3] units, as a convention, then if two variables have an engineering unit mismatch, then this match is not possible. Such criteria (c_d) could form a set C of disqualifying criteria. The weighting formula could then look as follows:

[3]International System of Units, from French: Système international (d'unités)

$$W_{i,j} = \begin{cases} \text{Match not possible}, & \text{if any } c_d(i,j) \text{ from } C \text{ failed} \\ \sum_q w_q * c_q(i,j), & \text{if all } c_d(i,j) \text{ from } C \text{ passed} \end{cases}$$

"Match not possible" from the formula can be a special value (for example 0) which is assigned to the weight $W_{i,j}$ to indicate that the match is not possible. The larger the set of disqualifying criteria C, the easier the matching, as presumably more variable combinations can be disqualified. On the other hand, it is important that we are certain that such matches are really not possible, because otherwise we would disqualify variable combinations which might be matches. This is why a good knowledge of the system is important, as well as good standardization. Designing the entire system from scratch (greenfield, Section 3.1), also has the benefit of being able to define a large disqualifying criteria set.

One issue to consider is when one variable has a certain criterion defined, and another not. For example, one variable might have its range set, and another not. In such cases, it is important to first look at other information, so for example, this range information might be in the type of the variable, as was given in the example above with a 16bit unsigned integer, and a 32bit signed integer where the ranges were defined to match the 16bit unsigned integer. In this particular example, the variables could be treated as a match, both in type, and in range, despite actually not matching either, as their combination provides a match for both. So, unless a strict standardization is applied when defining variables on both sides, having one side of the criterion defined, and another not, should not be treated as a criterion mismatch. On the other hand, it is not a match either, so the value of the criterion $c_q(i,j)$ needs to be defined somewhere in between. And, as was seen from the example, there also needs to be a possibility to consider other criteria, because the information needed for the criterion might be defined somewhere else.

There is also the question of setting the weights w_q for the individual criteria. The question how to set the weights for the criteria is not easily answered, and depends on the perceived importance of a particular criterion by the mapping unit designer, as well as possibility of the particular criterion mismatch despite variables matching, in a particular design. For example, the mapping unit designer knows that the variable types almost always match ("almost always" is not enough to be a disqualifying criterion), so variable type gets a high weight. On the other hand, the mapping unit designer knows that the range information is often mismatched (for example, as there was no coordination between the teams when determining the ranges), so this criterion gets a low weighting. The actual weighting values need to be manually determined by the mapping unit designer.

Finally, after the weights $W_{i,j}$ are set for all variable i and j combinations, the program needs to determine what is a match or not. One possibility would be to first consider high probability matches where the weights are within a certain range. These would then be checked, to see if there is only a single possible match within this (good) weight range. If there is, the variable combination can be declared a match. The reason for using a weight

range, is that, depending on the criteria, small fluctuations within the weight value might not mean that the variables are a less likely match. For example, one simulation variable has an almost perfect weight match with two automation system variables. The difference is that one matches the range, and the other matches engineering unit (take a look at Subsection 3.2.2 to see why engineering unit might not be a disqualifying criterion). Depending on the weights of the range and the engineering unit criteria, one might have slightly better weight than the other, but it is not directly clear which one is the match. Just making a guess is a bad strategy for the system, as it can introduce errors, and make the user's job more difficult, as one would then need to check the matches for possible errors. After the matches within that weight range are found, one can move to the next (worse) weight range and so on, until a certain threshold is reached. For all weights that are worse than the threshold, the program does not have enough information to declare them a match. For example, the variables might only have the type in common, and both might not have been matched in the weight range considerations below. It would probably be wrong to declare them a match, as only having the type in common is not enough to declare a match. At the end, after all possible variables are matched, the program should declare all matches found, as well as variables for which no match is found, to enable human intervention.

3.2.4 Requirements for Matching

Based on the information from this section, we can define a set of requirements for the matching procedure (RQ1-7).

RQ1: The matching should be based on the variable information available on the interfaces of the simulation and automation systems.

RQ2: The matching should happen automatically, matching as many variables as possible.

RQ3: The matching should be based on certain matching criteria, possibly including disqualifying criteria, as well as the criteria weights, and ultimately the variable combination weights, as described in Subsection 3.2.3.

RQ4: The matches should be determined in such a way that no wrong matches are produced, in other words, the program should stay on the "safe" side, as the program's mistakes might be difficult to find for the user.

RQ5: The program should have a possibility for manually defining which variables should not be matched, as both of the systems might have variables which have no equivalent in the other system (and are not relevant for matching).

RQ6: The program should provide the list of variables which were not automatically matched, as well as high probability matches for them, to facilitate the manual matching by the user. Ideally, the program should provide all relevant information

for the variables (for example the criteria), in order to make the manual matching by the user easier.

RQ7: The program must provide a possibility for the manual matches to override any decisions by the program, as well as to match the variables which the program was not able to match. If the program detects a low weight for one of the manual matches, the program might give a warning to the user.

These requirements for the matching aim to make the matching process clear and structured, as well as to enable a simple way for user to manually intervene, when needed.

CHAPTER 4

Proof of Concept

This chapter will describe a proof of concept implementation which was developed for the purposes of this thesis, and uses the techniques mentioned in Chapter 3. The two use cases are described afterwards, in Chapter 5.

4.1 Libraries and Assumptions

This section will describe the libraries which were used for the implementation of the proof of concept for this thesis, as well as the specific assumptions that the use cases must fulfill in order to interoperate correctly with the proof of concept implementation.

4.1.1 Used Libraries

The C programming language was chosen for the implementation of the proof of concept. Other language possibilities exist for both (open source) FMI and OPC UA libraries, such as: FMI4cpp[1] (C++), FMI4j[2] (Java), and PyFMI[3] (Python) for FMI[4]; and freeopcua[5] (C++), Eclipse IoT[6] (Java), and python-opcua[7] (Python) for OPC UA. This is certainly not a full list, and more libraries exist. In the end, the C language was chosen because of its universal applicability, portability, efficiency, and performance.

[1] `https://github.com/NTNU-IHB/FMI4cpp` ; accessed September 2021
[2] `https://github.com/NTNU-IHB/FMI4j` ; accessed September 2021
[3] `https://pypi.org/project/PyFMI/` ; accessed September 2021
[4] A good list of FMI libraries can be found at: `https://github.com/traversaro/awesome-fmi` ; accessed September 2021
[5] `https://github.com/FreeOpcUa/freeopcua` ; accessed September 2021
[6] `https://projects.eclipse.org/projects/iot` ; accessed September 2021
[7] `https://github.com/FreeOpcUa/python-opcua` ; accessed September 2021

FMI Library

For the FMI side, the FMI Library version 2.0.2 was chosen [8], which is an independent, and open-source implementation of the FMI open standard [Mod20]. The library was compiled and installed to a folder, which then needs to be given to the compiler with -I (which adds a directory to the list of directories searched for header files) and -L (which adds the path to the list of paths searched by the linker for archive libraries) flags, as can be seen in the proof of concept's makefile. No special options were used during the compilation. After these steps are taken, the library can then be included in the files, and used.

The FMU file needs to be provided to the program (and the library), in this case as a program argument. Additionally, a temporary folder must also be provided, where the FMU will be extracted. This additionally serves the purpose of extracting the FMU's XML file, which is needed for the matching. After initializing the library and the FMU, the simulation can be advanced step-wise with the following function: `fmi2_import_do_step()`. The function takes the FMU (as returned by the initialization), the last simulation time (the time at which the last simulation step ended, in the simulation), the current simulation step (length of the current simulation time), and a boolean variable which indicates if the last simulation step was performed or not. The values of the simulation variables can be obtained with the `fmi2_import_get_real()` function for real type variables, and similarly for others. The function requires the value reference, which along with the type uniquely identifies the corresponding variable. It is also possible to provide arrays as arguments, and read multiple variables at once. It is also possible to write to the simulation variables using `fmi2_import_set_real()` (and the correspondingly named functions for other types), in a similar way as the values are read. These three functions (and the corresponding type counterparts) are enough to run the simulation, after the initialization. The inherent step-wise simulation advancement means that step-wise interaction with the simulation is easy to implement. However, if the continuous time progression is required, despite the simulation advancement being step-wise, it is possible to give the impression of a continuously running simulation, as described in the Subsection 4.2.2.

OPC UA Library

For OPC UA, the open62541 library version 1.2 was chosen [9], which is an open source C99 implementation of OPC UA based solely on IEC 62541 [10]. The library allows compilation of the entire library into a single .c and .h file combination, which contains the whole library. This makes inclusions and portability easier. The library also offers the standard way of installing the libraries, however, the single file combination approach was chosen due to its simplicity, as well as to demonstrate its working. When compiling, the following two options need to be selected in cmake : first `"UA_ENABLE_AMALGAMATION"`

[8] `https://github.com/modelon-community/fmi-library` ; accessed April 2021
[9] `https://github.com/open62541/open62541/` ; accessed April 2021
[10] `https://open62541.org/` ; accessed April 2021

needs to be enabled - this compiles the library into a single .c and .h file combination, as discussed above; and second "UA_NAMESPACE_ZERO" needs to be chosen as "FULL" - this compiles all required nodes (instead of a reduced set) into the file, and is required for the proof of concept of this thesis.

After including the library in the program, the server needs to be initialized. There are functions then to include individual nodes to the server, as well as to set callbacks upon read/write of the values, which were then used in the proof of concept to "connect" the simulation and the OPC UA side. There are generally three ways to handle variable updating. The first and the simplest approach is to manually update variables, with no callbacks. The second approach (and the one which is chosen for the proof of concept), is to have "before-read" and "after-write" callbacks, which are called before the value is read, and after the value is written, respectively. The third way is using the so-called data sources. Using this, the values are not automatically updated, but instead upon reading, the callback provides a copy of the value, and the data source needs to provide its own memory management. [11]

A Python-based nodeset compiler is also provided with the library. It is used to generate a .c file (and the corresponding .h file) from the OPC UA nodeset file, with a function which generates all of the nodes for the OPC UA server. These .h file can then be included in the main program, and the function can be used. This is useful in order to prevent manual generation of all nodes. After all nodes are generated, they can afterwards be manually modified at will, as is done in the proof of concept. [12]

XML Library

For the XML parsing, the expat library was chosen [13]. The library enables easy parsing of the XML tags, which is what is primarily needed, using on-tag-start and on-tag-end callbacks. These callbacks have sufficed for most of the parsing in the program, apart from several special cases (for example reading units in the nodeset files, as they are outside of the tags). To this end, an additional function was used - the character handler. It loads entire XML segments (regardless of tagging), and enables reading the information between the tags. As the character handler is only needed in special situations, these are first detected by the tag callbacks, and the character handler then parses the value between the tags. The information within one file-parsing is transmitted using custom defined structs, which can then be used and shared between the tag callbacks, as well as the character handler.

[11] An overview of these approaches with some examples can be found at https://open62541.org/doc/current/tutorial_server_datasource.html (accessed April 2021).

[12] An introduction to the nodeset compiler can be found at https://open62541.org/doc/current/nodeset_compiler.html (accessed April 2021).

[13] https://libexpat.github.io/ ; accessed June 2021

4.1.2 Use Case Assumptions

Several assumptions had to be made in order to keep the implementation of the proof of concept within reasonable limits. First, all of the simulation-relevant OPC UA nodes should be in one namespace, the same one as defined in the server's .c file, and should not interfere with the namespace of the simulation relevant variables (variables for simulation in progress and manual mode, for example).

In order to perform the matching more easily, the units in the FMI and OPC UA variables (for those variables that have them) should match, i.e. be in the same physical unit representation. This would probably be difficult to achieve in a real example, as even FMI internally has different representation for the same unit (for example `"m/s2"` and `"m.s-2"`, both representations for $\frac{m}{s^2}$). The proof of concept implementation cannot handle different representations, and simply treats them as different units. This could be solved by converting all variables to a standardized representation, such as the UNECE codes[14].

For enumerations, it is assumed that the enumeration entries are listed only once in the FMI XML file. They are matched to UINT32 in OPC UA, in the program, however, the use cases do not need to match any enumerations.

For the units in the OPC UA nodeset files, it is assumed that there is only one unit definition with a single unit tag declaration. For the units as well as ranges in the nodeset file, it is additionally assumed that they follow the same tag structure as in the problem example nodeset file.

Structurally, it is assumed that all OPC UA value variables are directly below their parent object, in other words, variables that should share a parent, share the exact same parent, instead of being under an object that shares the parent with the object above the other variable. This does not mean that the entire structure must have the maximum of two layers, there can be as many layers as required, the assumption only states that there are no unnecessary layers. This is important for structural matching. Additionally, the parent node id must be specified within the `UAVariable` tag, for example like this: `<UAVariable SymbolicName="Inlet_Temperature" DataType="Double" ParentNodeId="ns=1;i=5003" NodeId="ns=1;i=6002" BrowseName="1:Inlet Temperature" AccessLevel="3">`. This ensures that the parent relations are read correctly within the proof of concept program.

4.2 Implementation

This section will describe the proof of concept implementation, developed for the purposes of this thesis. With regards to Chapter 3, the implementation is tailored to the problem variant where the simulation and the automation system are already defined, and a

[14] `https://unece.org/trade/cefact/UNLOCODE-Download` ; accessed August 2021

mapping unit design is needed to connect them. More details about the advantages as well as possible issues faced when using this problem variant can be found in Section 3.1.

4.2.1 Program Structure

There are three separate units in the proof of concept implementation. There is the emulation server - the main part of proof of concept, and the unit that performs the matching. It is called the emulation server because it hosts the OPC UA server, and provides the simulation data to OPC UA clients. In further text, the term OPC UA server will refer to the OPC UA server in particular, while the term emulation server will refer to the program, which facilitates the OPC UA server, the matching algorithm, and the interaction with the simulation (see the left-hand side of Figure 4.1). The second unit is the client, using automatic time progression, and the third unit is the client using the manual time progression. Time progression modes are described in their own subsection below (Subsection 4.2.2).

The client implementations are not extensive, and their purpose is only to show the working of the emulation server, in both of the modes. One could also use a separate, universal OPC UA client to interact with the emulation server, however this method was chosen as the library is already available, and it is an easier way to automate the tests. The emulation server consists of the server part, as well as of the matching segment, which is in its own .c file. The matching segment parses both of the FMI and OPC UA XML files (the paths to which are provided by the server part), as well as possibly the override file, and returns a viable matching, in the form of a matching table between OPC UA node ids and the FMI value references. The server part provides the OPC UA server functionality, as well as the interaction with the simulation. One additional part of the server is the function (and files) generated by the nodeset compiler. They enable easy creation of the entire OPC UA node tree, and are called by the server part, upon the creation of OPC UA server.

In terms of the system design and structure as described in Section 3.1, the mapping unit corresponds to the OPC UA server, as well as the matching sub-program where the matching is done. The client program represents the rest of the automation system, while the simulation environment is managed by the emulation server program. The correspondence can be graphically seen in Figure 4.1.

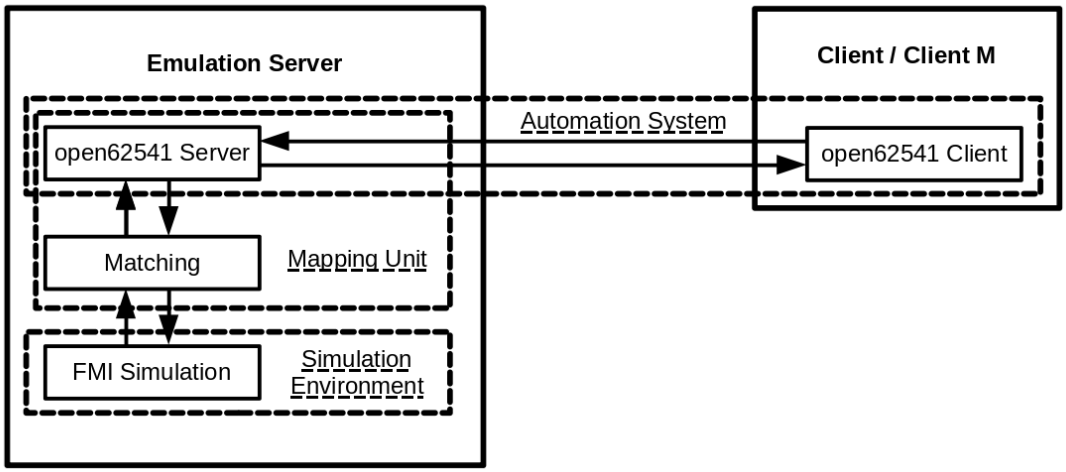

Figure 4.1: Program Structure Diagram
(Client M refers to the client with manual time progression)

4.2.2 Time Progression and Clients

As was already described in Subsection 4.1.1 (FMI), the time progression (the simulation advancement) with the FMI Library is step-wise. A function is called by the simulation master, in our case the emulation server application, which advances the simulation for a specified amount of time. This makes the manual mode client a natural fit for time progression. Using the manual mode, the emulation server does not advance the time automatically. Instead, the emulation server implements a method called `do_step()`, which takes a single float input, specifying how much the simulation should advance. In order to enable this time progression the `manual_mode` variable, which is also available to the clients over OPC UA, must be set to true. The client is free to modify and read the variables at will, and as many times as desired, the time only advances when the `do_step()` method is called. Contrary to this, one may wish for time to advance naturally, to at least partially simulate the interaction with a real world system. In this case, the client needs to turn off the manual mode, and set the `simulation_in_progress` variable to true (which was false during the manual mode), for as long as the simulation should run. The simulation can be paused by setting the variable to false again. The client can read and write the variables, however, time advances simultaneously with the wall time. There is nothing preventing the emulation server being modified in such way that the time is advancing faster or slower than the wall time. However, this was not implemented, as the manual mode gives enough control over the time, thus fulfilling the requirements for detailed analysis or fast time advancement. The simulation time can be checked in any mode by reading the variable `simulation_time`. It is also possible to mix the modes. So for example, if one wanted to see the wall time progression of the simulation after a week of running, one could use the manual mode

to advance the simulation with specified inputs at certain time points, for one week (which would be done by the program in hopefully much less time), and then switch the manual mode off, and start the automatic time progression. A short discussion about time progression can be found in Chapter 6.

Two examples are given below for how a client could read some value, two seconds into the simulation, using UML Sequence Diagrams. The first example, with automatic time progression, is given as Figure 4.2. The second example with manual time progression can be seen in Figure 4.3. In both diagrams, the sizes of the execution blocks (activation boxes) are not representative of the execution time, and should thus not be used to derive expected execution times for boxes where the execution time is not stated explicitly. Note that if the client is interested in the value after exactly two seconds have passed, the manual time progression is better, as it gives the value at the exact time point, while the automatic time progression is only approximately accurate in time.

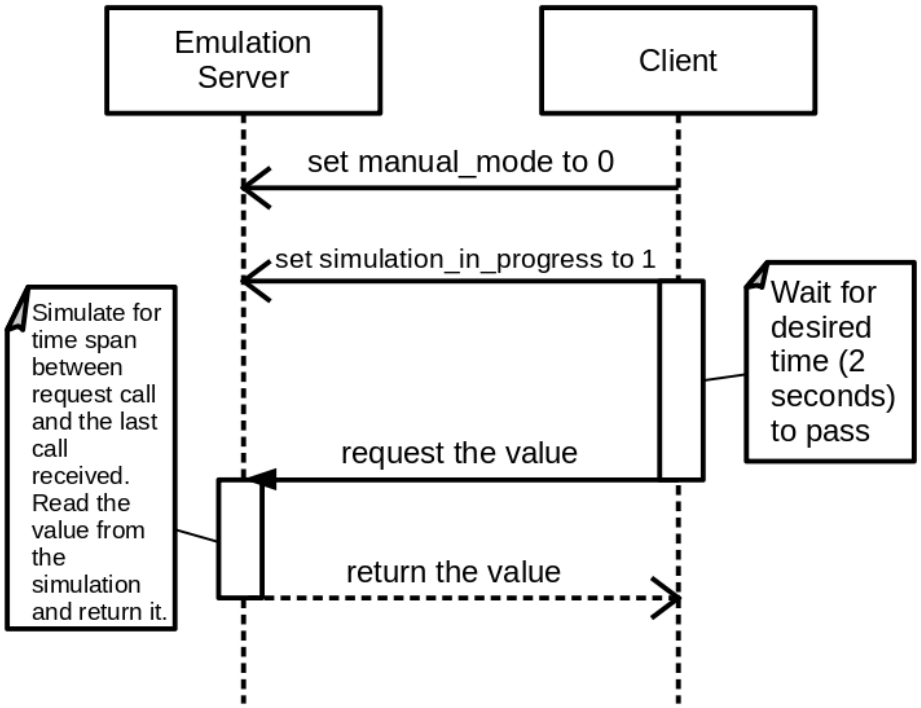

Figure 4.2: Automatic Time Progression: Value Read - UML Sequence Diagram

Implementation of the manual mode time progression is pretty straightforward. The OPC UA method `do_step()` calls the simulation function for time advancement, with the specified time as an argument, and the simulation then advances for the specified time. Alternative possibility would be to specify the time for which the simulation should advance as a separate variable, and then call the method to advance time without arguments. For the automatic time progression, the implementation is a bit more

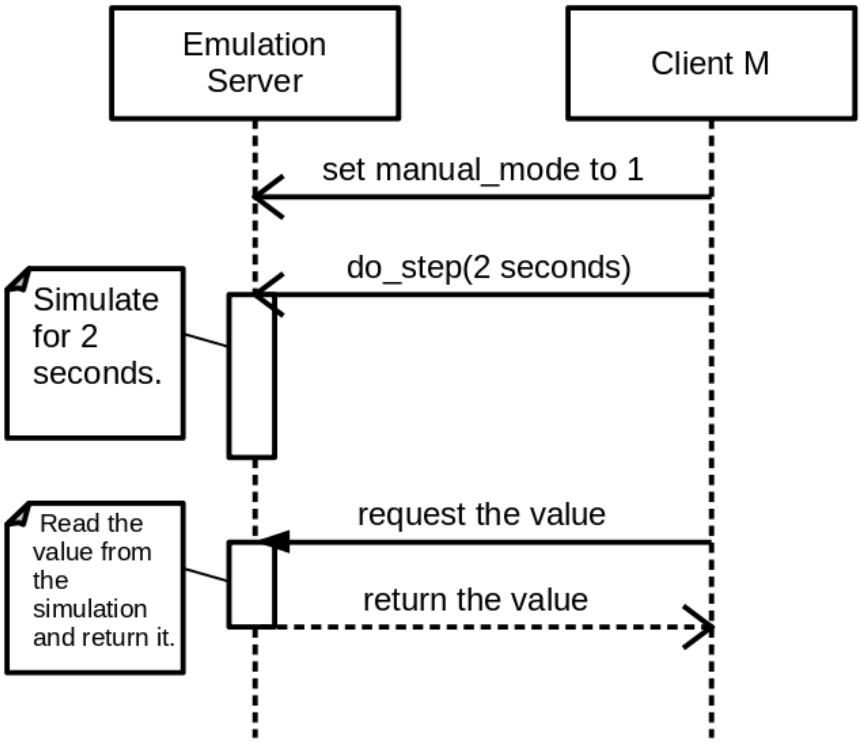

Figure 4.3: Manual Time Progression: Value Read - UML Sequence Diagram

complicated. As the natural time progression is not inherent to the underlying simulation, the emulation server had to use manual time progression, to give the impression of continuous time progression to the client. This is achieved using the before-read and after-write callbacks (as described in Subsection 4.1.1 - OPC UA), with the emulation server measuring time between OPC UA read and write calls. So for example, the simulation has started, after a minute, the client asks for a value of some variable. The emulation server then sees that the last call (interaction with the outside) was a minute ago (starting the simulation), and first advances the simulation for one minute. Then the emulation server reads the value from the simulation, and writes this value to the OPC UA node. The after-write call works in the similar fashion. Not to be confused by the naming, "after" refers to after writing the value in the OPC UA node, and it has nothing to do with the simulation. So after the value is written to the OPC UA node, the simulation is first advanced to the current time point, and then the new value is written to the simulation (nothing to do on the OPC UA side, as the client already wrote the value). This behavior gives the impression of natural time progression, despite the actual progression happening step-wise.

A problem that might arise from this implementation is the fact that if the simulation needs to advance for a long time (big time step), this might be quite time consuming. For

example if the clients have made no calls in the past hour, and then want two accesses within a second of each other; the first call might cause the emulation server to be busy with simulating the last hour, meaning that the answer will not come within reasonable response time. What is worse, the emulation server will probably still be busy with the simulation when the second call comes, thus also being unable to answer that query. This problem could be solved by advancing the simulation in regular intervals, by the emulation server, if no client calls are made. For example, the emulation server could advance the simulation every five minutes, if no client has made a call, in order to prevent long waiting time when a client call does come.

One problem that arose when dealing with the clients with natural time progression and the before-read callback, is that the emulation server has to write to the OPC UA variable at this time, thus also triggering the after-write callback. This happens when the client wants to read the value of a node, so the emulation server needs to first read the value from the simulation, and then write the newly read value to the OPC UA node (triggering the after-write callback). It is important that we distinguish between the server-side triggered after-write callbacks, and the actual writes from the clients. When a client makes a write, the simulation is first advanced to the current time, and then the desired value is written to the simulation. In the case of a read, the value is provided to the client by the emulation server, and then the after-write is triggered, which advances the simulation for a short time (the time between the calls), however it would then write this value to the simulation. This can lead to problems. For example if the value was changed by the simulation in the meantime, it would lead to incorrect results. To this end, the emulation server recognizes its own calls using session ids (with the assumption that the emulation server session id does not change during the program execution). If there is a server-triggered after-write callback, after the server is started (open for clients), the emulation server will simply skip the writing to simulation step, as such call can only come from the write in the before-read callback (the emulation server performs no self-initiated writes), and writing to the simulation value in this case would be incorrect, as described above.

4.2.3 The Emulation Server

Much of the emulation server's interaction with the clients was described in the previous subsection. However, before the OPC UA nodes are ready for the clients, several steps need to be taken. First of all, the emulation server initializes the FMU, path to which was provided as a command line argument. The library extracts the FMU to a given temporary folder, readying it for use. Then the emulation server calls on the matching segment of the program to provide the matching, as well as the information on OPC UA variables that will be hosted. The matching itself is described in the Subsection 4.2.4. The emulation server then calls on the automatically generated (by the nodeset compiler) function which creates all of the nodes for the OPC UA. The emulation server then sets callbacks for the particular nodes of interest (those [non-abstract variables] that have a matching, i.e. a counterpart in the simulation), in order to enable correct interaction with the simulation.

After that, the variables which are provided to OPC UA clients to interact with the simulation are declared. These are: `simulation_in_progress`, `manual_mode`, and `do_step()` - the functionality of which was described in the previous subsection, and the `simulation_time` which provides the clients with easy access to read the current simulation time. The last step is then to start the OPC UA server. After the emulation server is terminated, it performs some clean-ups of the FMU, OPC UA, as well as internal variables, and exits the program.

4.2.4 Matching

The matching part of the program receives the paths to the FMI and OPC UA XML files, as well as possibly to the override file, from the main part of the program, which received the paths as a command line argument. For both FMI and OPC UA, the parsing works as follows: first there is a pre-parsing which counts the number of variables, units, and other useful information, which is used to properly initialize the memory needed for the rest of the parsing operation. Second, the files are parsed, now filling the memory with the information from the files (i.e. reading the XML and converting the information to an internal representation). Alternatively, if one desires to avoid double parsing, a good solution would be to use a dynamic data structure, such as a linked list to store the parsed information, and with that, parse the file only once. Lastly, some special steps are required to finalize the parsing. After these steps are taken for both FMI and OPC UA files, the actual matching is performed using the prepared information about both FMI and OPC UA variables. The matching results in a table of matched pairs of OPC UA node ids, and FMI value references. Lastly the used memory which is no longer needed is freed.

FMI Parsing

The FMI pre-parser determines the number of variables and the number of types. This information is used to declare the memory required for the rest of the program. The parser then fills in the information in internal representation. For the variables, this consists of the name of the variable, the value reference, and the causality (input, output, or parameter/other). Note that all variables that are neither input nor output variables in causality will be marked by the program as parameter, for simplicity. The causality is currently not used for matching. The type should also be defined here, and is written to the variable information. The unit and the min/max values may be defined directly here at the variable, or they may carry a reference by the unit key. In case of the direct definition, they are written to the variable information, and this concludes the modification needed for this variable. In the case that only a reference is provided, the unit key is written to the variable, which will then be used later to find the unit. Along with the variables, the XML file can also declare units which will be used indirectly (through references). All the information for these units is filled, along with the unit key, which will then be used to find the variables which have this unit. The enumerations are declared as types. At this point the program also counts the number of items in the

enumeration, however this is currently not used for matching. After the file is parsed, the units can be filled. This means that the program iterates through the variables and units, finds the matching keys, and copies the unit and the min/max values (if present) over. The variables are thus complete with units. The final step needs to be taken is to reduce the unnecessary number of variables. Some variables are defined as aliases of others, under a different name, but with the same value reference and type. FMI standard declares that such variables must have the same value, thus they are duplicates. The program detects such duplicates, and prunes them from the list of variables for the matching. If they have different limits, then the strictest limits are taken, in accordance with the FMI standard.

OPC UA Parsing

The OPC UA XML file parsing is somewhat more complex than the FMI parsing. It starts with the preparsing, where the program records the number of variables, engineering unit definitions, variable range definitions, as well as the total number of nodes. During the parsing the following information is also recorded, for each node: its id, parent id, and if it is abstract or not. For variables additionally the following information is stored: type information, the browse name, and the access level. For the engineering units and the value range definitions, there is a chain of several XML tags that need to be followed, in order to get to the value. The value is then read by the character parser (as it is not a tag), as described in Subsection 4.1.1 (XML), and stored in separate arrays. After this parsing is done, the program then fills the units and ranges to corresponding variables. To this end, as the parent node ids are already recorded for each unit and range, the program iterates through the variable array, and finds the corresponding parent, to which a value can then be attached. Then, the abstract variables need to be removed (the non-abstract are extracted). The abstract property is propagated through the tree, top-down, and then prune the entire abstract branch. The abstract nodes have no counterpart in the simulation. In the program, extracting the non-abstract variables is done before the unit and range fill, in order not to waste time on units and ranges of abstract nodes.

Applied Matching Techniques

After the parsing is done, the matching can be started. With respect to Subsection 3.2.3, the following decisions were made: there is one disqualifying criterion, the data type. The other three criteria: the range, the engineering unit, and the structure all have the weight of 1. The ranges are considered when checking the data type, whereby for example for FMI REAL, if no range is defined, a match with an OPC UA DOUBLE is more likely than a match with an OPC UA FLOAT (weight penalty of 1). Otherwise, having ranges defined on one side, and not on the other is penalized. On the other hand, having the engineering unit defined on one side and not on the other is penalized with the weight of 1 (i.e. it is treated as a mismatch). For the structural matching, if the variables do not belong to the same structure, they are penalized with 1 weight. The weighting is inverse, meaning that the smaller weights mean the match is more likely, with the value

of 1 being the best, and with mismatches adding additional weight (penalization). The exact penalties for each variable combination can be seen in the Appendix.

The internal table for the matching is declared as follows: all FMI variables on one axis, and all OPC UA variables on the other. This matrix stores integer values, and it will be modified throughout the matching process. The integer values have the following meaning: 0 - match impossible, 1 - match likely, 2 - match less likely; and so on, each higher number indicating a lower likelihood of a match (and 0 as a special value designating that the match is not possible).

The matching algorithm can be seen in a UML Activity Diagram in Figure 4.4, and it is explained in detail in the text below.

First step is to perform the type assignment. The possibility of each of the OPC UA types for each of the FMI variables is considered here. For this, the FMI type, as well as the range (if present) are taken into consideration. For example, an FMI Real variable gets the value of "1" (according to the scale above) to the following OPC UA types: Float, Double, Duration. Additionally, if the max value is defined, and it is less or equal to Float Max, the matrix entry for Double and Duration goes to "2" (as these are both in the long float format). The rest of the FMI variables are handled in a similar fashion, and thus have possible types assigned. The results are then copied over to the original matching table, assigning the values in accordance with the type of the OPC UA variable. After this is done, the program needs to check the min and max values. If min/max is defined in both the FMI and the OPC UA variable, and the value is different, the probability value of 2 is assigned (if it was 1). The reason for not increasing the probability level (decrease presumed probability) for the variables where one side has the min/max defined, and other one does not, is that this information might already be contained within the type (as is the case in the Float example above).

The next step is to check the units. If the units are defined for both variables (matchable FMI and OPC UA), and they differ, the match is less probable (increase the value in the matrix). The reason that it is not impossible, is that some units have different representations, for example on the FMI side with might see "degC" as the unit, and on the OPC UA side "°C". This is assumed not to be the case in the chosen problem, as it additionally complicates the matching.

Next the override file needs to be checked. If the override file is present, the program goes through the OPC UA variables and their overrides. If a variable is overridden with "−1", it should not be matched (i.e. a variable that has no simulation counterpart, and is designated as such by the user), and all of its values in the matching matrix are set to zero. If the match is overridden with a valid FMI value reference, the rest of the entries in the matrix for that OPC UA variable need to be replaced with zeros. A special care is taken here in order to only let the correct FMI variable have the non-zero value, as one FMI value reference might appear multiple times, if the FMI variables are of different types. At this point, the structural matching takes place. If there is a match (through override), we record the structure of the matched FMI variable (the highest

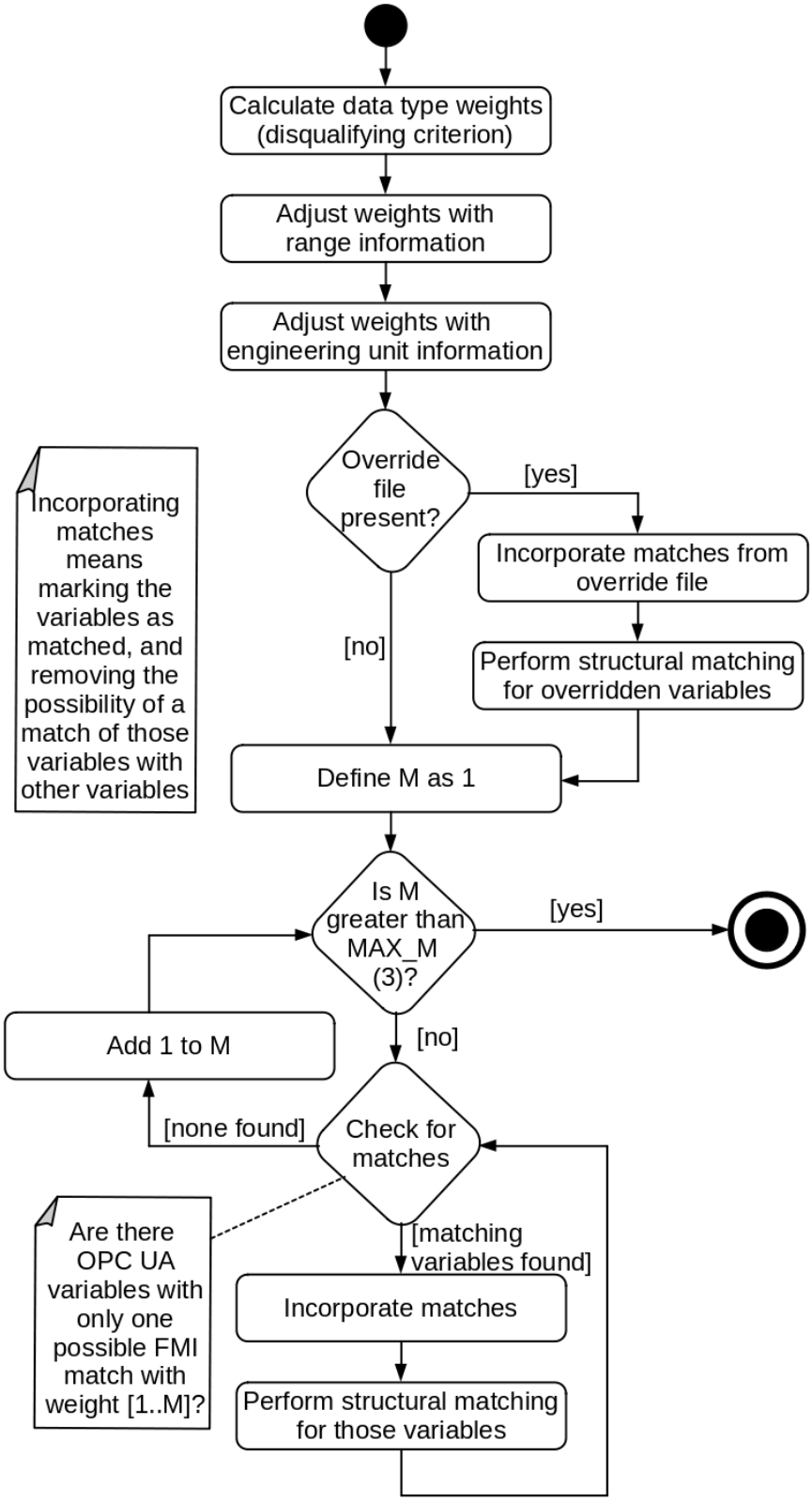

Figure 4.4: Matching Algorithm UML Activity Diagram

group level, so for "robot.arm1.centerOfMass", the structure would be "robot.arm1"), and the structure of the matched OPC UA variable (parent node id). All combinations of FMI variables (on one side) and OPC UA variables (on the other) are checked. If both have the same structures as the ones in the match that was overridden, they are considered structurally matching, and are not penalized. If one of the variables coincides in structure with the matched variables, but the other one does not, they are considered structurally mismatching, and are thus penalized with the weight of 1. If neither of the structures coincide with the matched variables no action is taken.

After override is done, the possible matches need to be checked. To do this, the program first iterates though the entire array, for each OPC UA variable, and observes if the variable only has one possible match, of certain level(s) (weight(s)). First a check is performed to see if there are variables with only one level 1 match, these are counted as matches, and they are added to the matching array which will be returned to the main program. The possibility of a match for that FMI variable for all other OPC UA variables is then removed. If a match is found, structural matching is performed again (same as for the overridden variable matches) for the matched pair. The list of the OPC UA variables is then iterated again, this is done as long as there are changes to the matching table. The reason for multiple iterations, is that there might be an OPC UA variable with one possible FMI match, and one with two possible FMI matches, one of which was shared with the first OPC UA variable. In this case, in the first iteration, only the one which has a single possible match will be matched, however, after this particular FMI variable is assigned to an OPC UA variable, the first OPC UA variable is left only with a single possible match, so it can also be assigned a match.

After this matching is performed for all level 1 matches, the program then checks level 2 matches (structural matching is performed again in case of matches). If there is an OPC UA variable with a single level 2 match (and no level 1 matches), it is assigned as matched, and do the same steps as described above. This check is also repeated until there are no more changes. Lastly, level 3 matches are also checked in a similar way. Level 3 is considered a cut-off in this configuration, and any variables with higher levels (lower probability) are considered not matchable.

With this, the matching process is done. The matching array which is returned is a single-dimension array which consists of the OPC UA node id for the match in the $2k$ position, and the corresponding FMI value reference in the $2k + 1$ position. The OPC UA variables for which there was no match found, as well as those explicitly defined as without a match in the override file, have a value of -1 as their match. This special value is then checked by the server part of the program, and if such value is detected, no interaction with the simulation variables is performed for that OPC UA variable.

Assuming that the number of variables for both FMI and OPC UA is n, the total asymptotic complexity of the matching algorithm is $O(n^3)$. The preparations, initializations, and weight adjustments before the matching itself have the complexity of $O(n^2)$. In the worst possible case, the matching algorithm will make one match within each loop iteration (loop being designed by the "Check for matches" condition in Figure 4.4), thus

necessitating $O(n)$ iterations. Having one match per loop is the worst possible case, as having more matches would reduce the number of total iterations, while having zero matches would stop the loop. In each of those iterations, the program iterates over the matching table and counts the possible matches ($O(n^2)$). This gives the complexity of $O(n^3)$. Additionally, once for each match (so $O(n)$ times), structural matching is performed, which costs $O(n^2)$, as the program has to go through the entire matching table to adjust the weights; this also gives $O(n^3)$. So the total asymptotic complexity of the matching algorithm is $O(n^3)$.
The chosen number of (weight) levels that are acceptable for the matching to be possible does not change the complexity of the algorithm. The number of possible total loop iterations cannot increase - the variables that are matched previously are skipped in further levels, while the variables that were not matched before couldn't contribute to the number of loop iterations previously.

4.2.5 Compilation and Program Execution

The requirements for the libraries and their versions are described in Subsection 4.1.1. A diagram for the entire compilation process and the required files can be seen in the Figure 4.5. Before the compilation, first the nodeset compiler has to be called in order to generate the required .c and .h files, containing the function that will generate the node tree in the OPC UA server. The files and the function have to have the following names respectively: `myNS.c`, `myNS.h`, and `myNS()`. The nodeset file has to be in the same folder as the program, as well as the FMU, and the temporary folder for extracting the FMU. The program itself will extract the FMU to the temporary folder and read the model description file from there. The program can then be compiled (gcc version 9.3.0 was used). One can run the emulation server with the arguments corresponding to the FMU file path, the temporary directory where the FMU will be extracted, then the OPC UA nodeset file, and finally the optional override text file. All paths have to be full absolute paths. The override file has to have the format readable by the program, meaning that each line contains two entries, first the OPC UA node id, and then the corresponding FMI value reference (or "−1" to indicate that this OPC UA variable should not be matched). The emulation server runs a single simulation, meaning that two clients cannot use different modes (manual and automatic time progression) at the same time. In addition, two C libraries that are also used but not on the diagram (Figure 4.5) are dynamic linking library and the realtime extensions library.

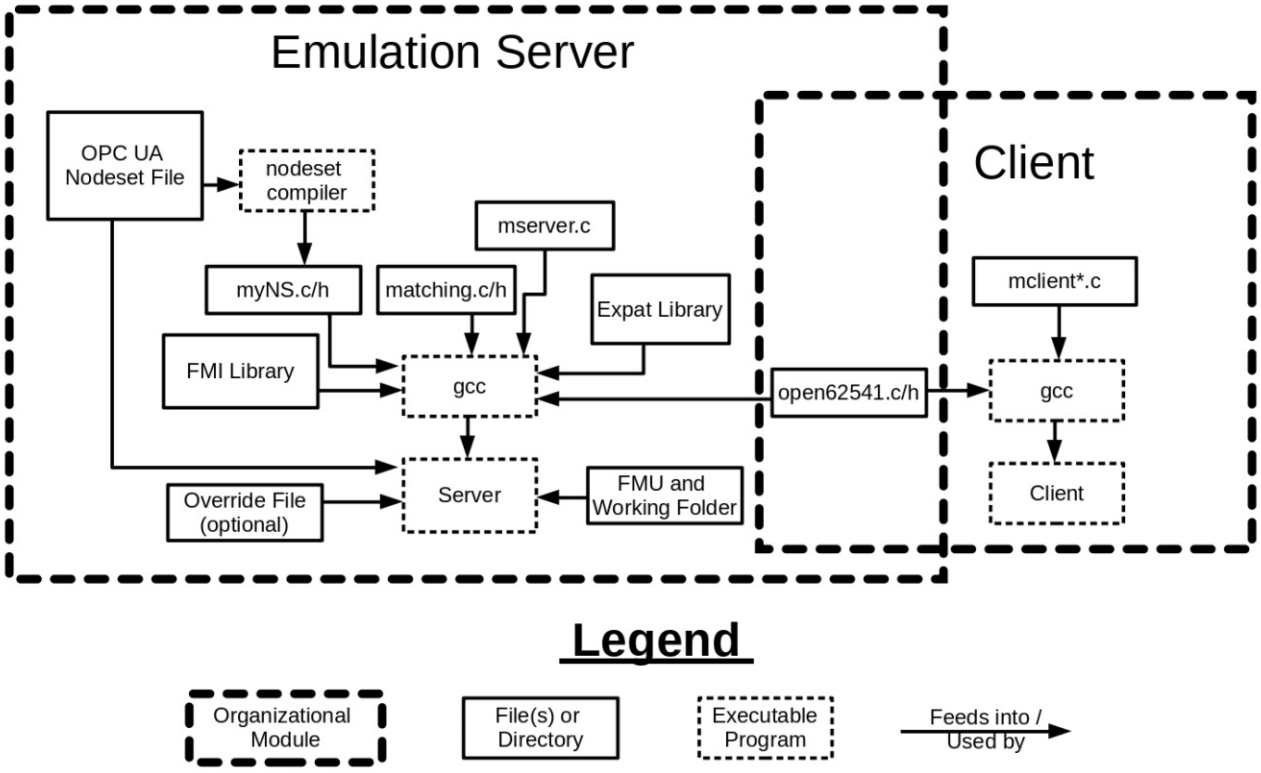

Figure 4.5: Compilation and required files diagram

CHAPTER 5

Use Cases

This chapter describes the two use cases, which were used along with the proof of concept implementation of this thesis to demonstrate its functionality. The first use case is a simple physical free fall experiment, and it mainly serves as a demonstration of matching capabilities. The second use case is a fluid heat flow experiment, which demonstrates the data flow capabilities after the matching.

5.1 Use Case I - Free Fall

The first use case chosen for the proof of concept is relatively simple, a body performing a free fall in a friction-less environment. The physics diagram for this use case can be seen in the Figure 5.1.

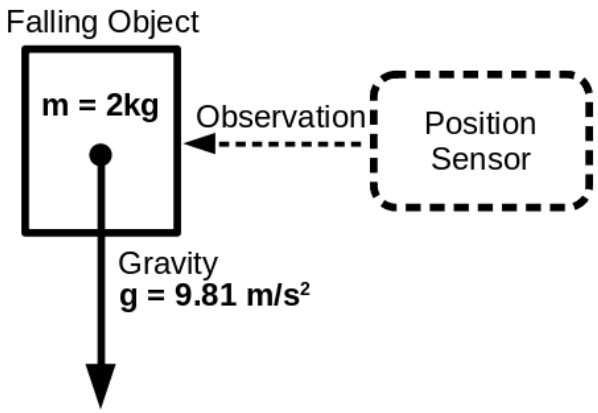

Figure 5.1: Use case I: free fall physics diagram

5.1.1 Simulation Model

The chosen use case is a free fall of a 2kg body in a friction-less environment. The FMU was generated using the OpenModelica program (version 1.16.5), and the diagram of the use case within the program can be seen in Figure 5.2. Despite the simplicity of the problem, there are still 27 variables before the duplicate pruning, and 23 afterwards. The variables after duplicate pruning can be seen in (the left part of) Table 5.1, in Subsection 5.1.3. There are multiple variables describing similar physical entities, however they are not aliases in the FMU.

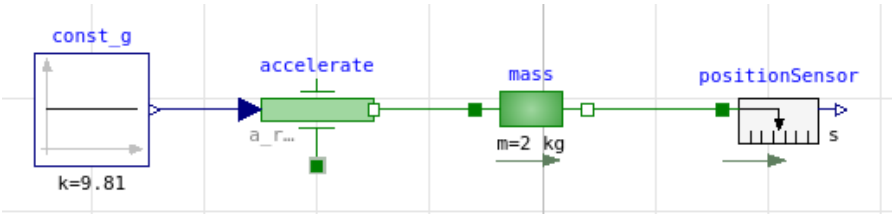

Figure 5.2: Use Case I as designed within the OpenModelica program

5.1.2 Automation System Information Model

An OPC UA nodeset file was created using the UAModeler program[1]. The information model contains the following variables: `Acceleration`, `Velocity`, `Position`, and `Mass` of the falling mass object. As this information model was manually designed, there are no duplicates, in contrast to the simulation model. However, there are abstract variables, which do not partake in matching. Similarly to the duplicate FMI variables, these abstract OPC UA variables are pruned before matching. There are also additional variables that do not correspond to simulation variables, these were not matched either. These variables can be seen in (the right part of) Table 5.1, in Subsection 5.1.3. In addition to these information model variables, which are defined in the nodeset file, there are three additional variables and a function under the `sim_control` object, which correspond to the variables used for simulation control. The functionality of simulation control variables and the function is described in Subsection 4.2.2. Figure 5.3 shows both the information model (nodeset) variables (left hand side), as well as the simulation control variables (right hand side).

[1] `https://www.unified-automation.com/products/development-tools/uamodeler.html` ; accessed April 2021

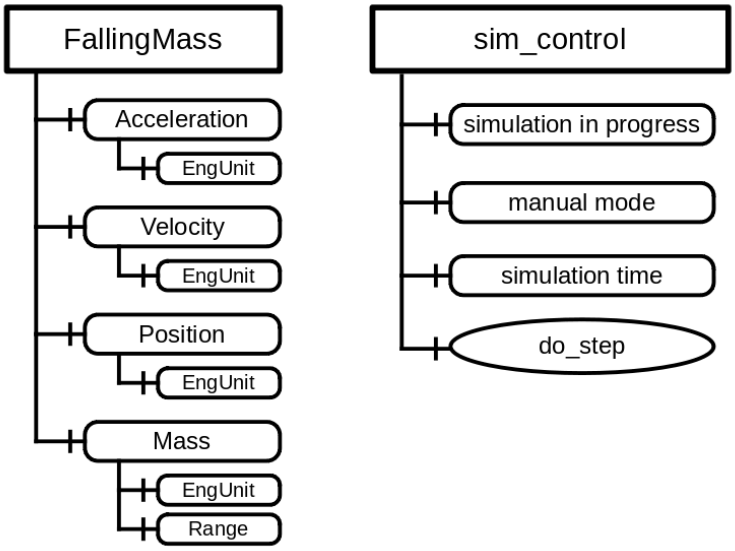

Figure 5.3: Use Case I: OPC UA Information Model Instances

5.1.3 Matching Procedure for Use Case I

For the chosen use case, the proof of concept implementation recognizes 27 FMI variables, out of which 23 are originals, and 4 are duplicated; as well as 15 OPC UA variables out of which 4 are abstract, and 7 are not relevant for the simulation, leaving four variables to be matched. Matching was performed based on variable types, ranges, engineering units, and structure, as described in the Subsections 3.2.2 and 4.2.4. The table of matches is given as Table 5.1. The table columns represent the variable name, variable type, and variable unit ("/" for no unit) for both FMI and OPC UA, as well how the variables were matched (abbreviated). The table rows are the variables, with always the FMI variable and its OPC UA match in the same row.

Table 5.1: Table of Matches for the Use Case I;
Match Column: **Imm**=Immediate, **Man**=Manual Override, *NM*=Not Matched

FMI Variable Name	FMI Type	FMI Unit	Match	OPC UA Unit	OPC UA Type	OPC UA Variable Name
accelerate.a	Real	m/s2	*NM*			
accelerate.a_ref	Real	m/s2	*NM*			
accelerate.s_support	Real	m	*NM*			
accelerate.flange.f	Real	N	*NM*			
accelerate.s	Real	m	*NM*			
accelerate.v	Real	m/s	*NM*			
der(accelerate.s)	Real	m.s-1	*NM*			
der(accelerate.v)	Real	m.s-2	*NM*			
accelerate.useSupport	Boolean	/	*NM*			
const_g.k	Real	/	*NM*			
const_g.y	Real	/	*NM*			
positionSensor.flange.f	Real	N	*NM*			
positionSensor.s	Real	m	*NM*			
mass.a	Real	m/s2	**Imm**	m/s2	Double	Acceleration
mass.flange_b.f	Real	N	*NM*			
mass.s	Real	m	**Man**	m	Double	Position
mass.v	Real	m/s	**Imm**	m/s	Double	Velocity
mass.L	Real	m	*NM*			
mass.m	Real (min 0)	kg	**Imm**	kg	Double (min 0)	Mass
mass.flange_a.f	Real	N	*NM*			
der(mass.s)	Real	/	*NM*			
der(mass.v)	Real	/	*NM*			
mass.stateSelect	Enum	/	*NM*			
			NM	/	Boolean	IsNamespaceSubset
			NM	/	DateTime	Namespace PublicationDate
			NM	/	String	NamespaceUri
			NM	/	String	NamespaceVersion
			NM	/	BaseData Type	StaticNodeIdTypes
			NM	/	BaseData Type	StaticNumeric NodeId Range
			NM	/	String	StaticStringNodeId Pattern

The FMI variables are structured into the following four groups: `accelerate`, `const_g`, `positionSensor`, and `mass`. There are four relevant OPC UA variables, but there is also an additional set of variables which are not specific to the use case, and are not matched (last 7 variables). These 7 OPC UA variables have access level zero, and are ignored by the program based on that. For the matchable variables, the FMI variable is given on the left, and the OPC UA variable on the right. How a match is made is given in the column "Match", whereby "Immediate" means that the program found this match without any help from the override file, and "Manual Override" means that this variable was overridden in the override file (matched manually).

The matching procedure then works as follows: after running the program without an override file specified, the matches which are marked with "Immediate" match in the table are found. The program provides likely matches for other variables. In the example, from the four OPC UA variables that should be matched, only the `Position` variable is not immediately matched. Program's suggestions can be seen in Figure 5.4.

```
The following matches [OPCUA] could not be matched, high probability matches listed below:
NodeID: [ns=1;i=6040] BrowseName: [1:Position] AccessLevel : [3] Type: [Double] Unit: [m]
    Name: [mass.s] Vref: [10] Causality: [parameter] Type: [Real] Unit: [m]
    Name: [mass.L] Vref: [18] Causality: [parameter] Type: [Real] Unit: [m]
```

Figure 5.4: Program's suggestions for matching the `Position` variable

The program does not know which FMI variable the OPC UA `Position` variable corresponds to, and lists all (two) high probability (level 1) matches that it found. The provided possible matches help the user define the override file, in the example above, the user could have deduced which match was correct based on the variable names.

In particular, the matching for the use case works as follows: after type, range, and engineering unit matching is done, there are several level 1 matches. For example, the OPC UA `Acceleration` variable has a level one match with the following FMI variables: `accelerate.a`, `accelerate.a_ref`, and `mass.a`. The program cannot deduce which of these variables `Acceleration` corresponds to, so it does not make further progress here. `Velocity` and `Position` variables have similar results. The `Mass` variable can however immediately be matched, as it has only one level 1 match, namely `mass.m`. This is the only level 1 match as it is the only variable with the engineering unit `kg`, as well as being the only one with a minimum limitation of 0. After this is matched, the structural matching takes place. The FMI structure of the match is `mass`, and the OPC UA structure is the simulation variables object. So in the case of `Acceleration`, the FMI variables `accelerate.a` and `accelerate.a_ref` get a weight penalty of 1, as they are in the `accelerate` structure. This happens for other variables as well. After these penalties are applied, both `Acceleration` and `Velocity` are matched, as there is only one level 1 matching left for each of them. The only variable that is left is the `Position`, and the program cannot deduce if it should be matched to `mass.L` or `mass.s`, as they are both Real variables in meters (`m`), belonging to the `mass` FMI structure. To progress further, human intervention is required. The human

must provide the matching variables in the override file, in this case OPC UA `Position`, and FMI `mass.s` (by node id and value reference, respectively). After running the program with the override file, all matches can be made (`Mass`, `Acceleration`, and `Velocity` automatically, and `Position` manually), and the matching is thus complete (Table 5.1). The data flow is established (variables can be read by OPC UA clients; in particular the `Position` variable is read by the client examples).

5.2 Use Case II - Fluid Heat Flow

The aim of this use case is to demonstrate the workings of the data flow capabilities of the matching unit, and not the semi-automatic matching capabilities. The physical experiment works as follows: there is a coolant flowing through a pipe, the source temperature of the coolant is 20 degrees Celsius. A heat source is heating the coolant through a thermal conductor. There is a pump which determines the volume of the flow through the pipe (in $\frac{m^3}{s}$). This volume flow can be controlled, as in an input to the system. The inlet coolant temperature as well as the outlet coolant temperature can be read, and are considered information of interest for this use case. The physics diagram for this use case can be seen in the Figure 5.5.

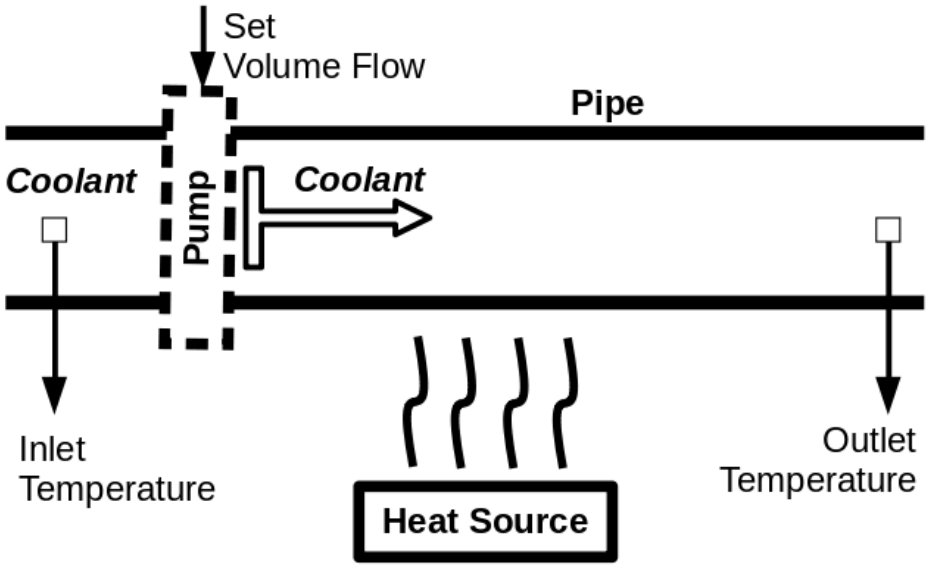

Figure 5.5: Use case II: fluid heat flow physics diagram

5.2.1 Simulation Model

The FMU was generated using the OpenModelica program (version 1.16.5), and the diagram of the use case within the program can be seen in Figure 5.6. The simulation

model is an adapted version of the SimpleCooling example from the Modelica library (under Thermal: FluidHeatFlow). The simulation model for this use case is very similar to the mentioned example, the only difference being, that the desired volume flow of the pump is an input, and can be changed. There are a total of 175 variables, 154 of which are left after duplicate pruning. Due to the sheer number of variables, they will not be listed here, but they will be discussed in Subsection 5.2.3.

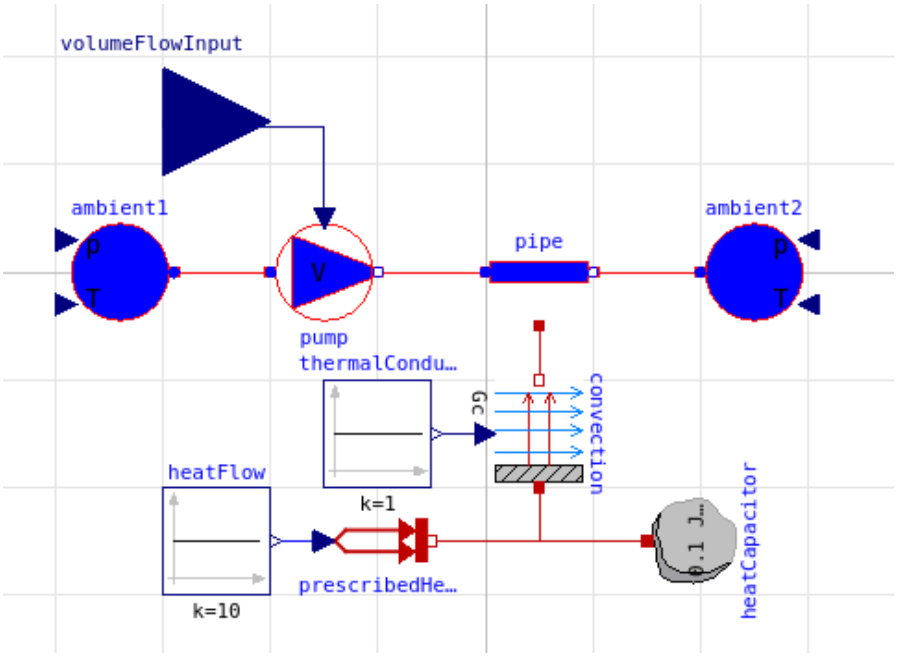

Figure 5.6: Use Case II as designed within the OpenModelica program

5.2.2 Automation System Information Model

An OPC UA nodeset file was created using the UAModeler program. The information model contains two objects: `Pump` and `Pipe`. The `Pump` object has a sole variable `Volume Flow`, which serves as an input to the simulation. The `Pipe` object has two variables: `Inlet Temperature` and `Outlet Temperature`, which can be used to check the inlet and outlet temperatures, respectively. Same as with Use Case I, Subsection 5.1.2, in addition to the information model variables from the nodeset file, there are three additional variables and a function under the `sim_control` object, which correspond to the variables used for simulation control. The functionality of simulation control variables and the function is described in Subsection 4.2.2. Figure 5.7 shows both the information model (nodeset) variables (left hand side), as well as the simulation control variables (right hand side).

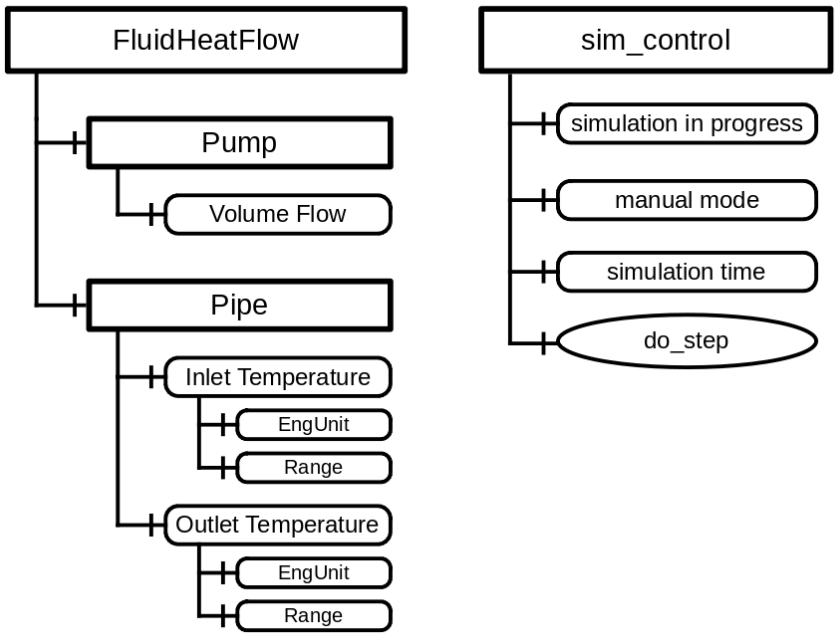

Figure 5.7: Use Case II: OPC UA Information Model Instances

5.2.3 Matching Procedure for Use Case II

This use case presents a much more difficult matching than the first use case. The reason for this, is that there are many more FMI variables (154 after duplicate pruning), and many of them have the same or similar attributes (for example, there are many temperature variables). The structural matching is not helpful either, as all three variables that need to be matched, from the OPC UA information model, are from different objects within the FMU (namely: none, `ambient1`, and `ambient2`; see FMI variable names in Table 5.2).

The best possibility for making a match is for the `Volume Flow` variable. Despite the volume flow being expressed in $\frac{m^3}{s}$, the input variable itself in the FMU has no unit (as it is simply a real value input). OPC UA information model was designed to reflect this. Looking for other variables with no units, and that adhere to the type criteria, the program gives the following suggestions for the `Volume Flow` variable (Figure 5.8):

```
NodeID: [ns=1;i=6001] BrowseName: [1:Volume Flow] AccessLevel : [3] Type: [Double]
    Name: [volumeFlowInput] Vref: [39] Causality: [input] Type: [Real] No unit
    Name: [heatFlow.k] Vref: [77] Causality: [parameter] Type: [Real] No unit
    Name: [heatFlow.y] Vref: [78] Causality: [parameter] Type: [Real] No unit
    Name: [thermalConductance.k] Vref: [135] Causality: [parameter] Type: [Real] No unit
    Name: [thermalConductance.y] Vref: [136] Causality: [parameter] Type: [Real] No unit
```

Figure 5.8: Program's suggestions for matching the `Volume Flow` variable

As there are only five variables suggested by the program, the list is quite manageable. Just based on the variable names, a human operator could easily deduce the match (`volumeFlowInput`).

For the other two variables, the `Inlet Temperature` and `Outlet Temperature`, the program lists many possible variables as good (level 1) matches. In particular, the same 27 variables are listed as good matches for each of the variables. Determining which variables are the matches in this case is not so simple, and requires a good knowledge of the FMI model.

As the program is not capable of matching any of the variables on its own, they must all be specified in an override file. Even after matching some of variables manually, the structural matching cannot help in this case, as there are many FMI variables in total, and all variables that should be matched belong to different structures. The table of matches is given as Table 5.2, and only contains OPC UA variables and their matches (the rest of the FMI variables are excluded, for readability).

Table 5.2: Table of Matches for the Use Case II;
Match Column: **Man**=Manual Override

FMI Variable Name	FMI Type	FMI Unit	Match	OPC UA Unit	OPC UA Type	OPC UA Variable Name
volumeFlowInput	Real	/	**Man**	/	Double	Volume Flow
ambient1.T_port	Real (min 0)	K	**Man**	K	Double (min 0)	Inlet Temperature
ambient2.T_port	Real (min 0)	K	**Man**	K	Double (min 0)	Outlet Temperature

5.2.4 Data Flow

This use case has two clients: manual and automatic time progression, similarly to the Use Case 1. However, in this Use Case, they perform different tasks, in order to show the functionality of the data flow. The client with manual time progression functions on a simple time-triggered principle, while the automatic time progression client functions on the basis of temperature control.

Time-Triggered Client

The manual time progression client works as follows: volume flow starts at $0.1\frac{m^3}{s}$, after a certain time period (at $2.2s$ simulation time), the client starts increasing the volume flow of the pump by $0.05\frac{m^3}{s}$, every $0.05s$, until it reaches $1\frac{m^3}{s}$. This results in the following behavior: at the start, the temperature increase is slightly delayed, due to the processes which need to transpire (heat flow, convection, etc.). The temperature then rises, until enough coolant passes through that the temperature starts falling again. The temperature then stabilizes at 30 degrees Celsius. This behavior can be seen in Figure 5.9.

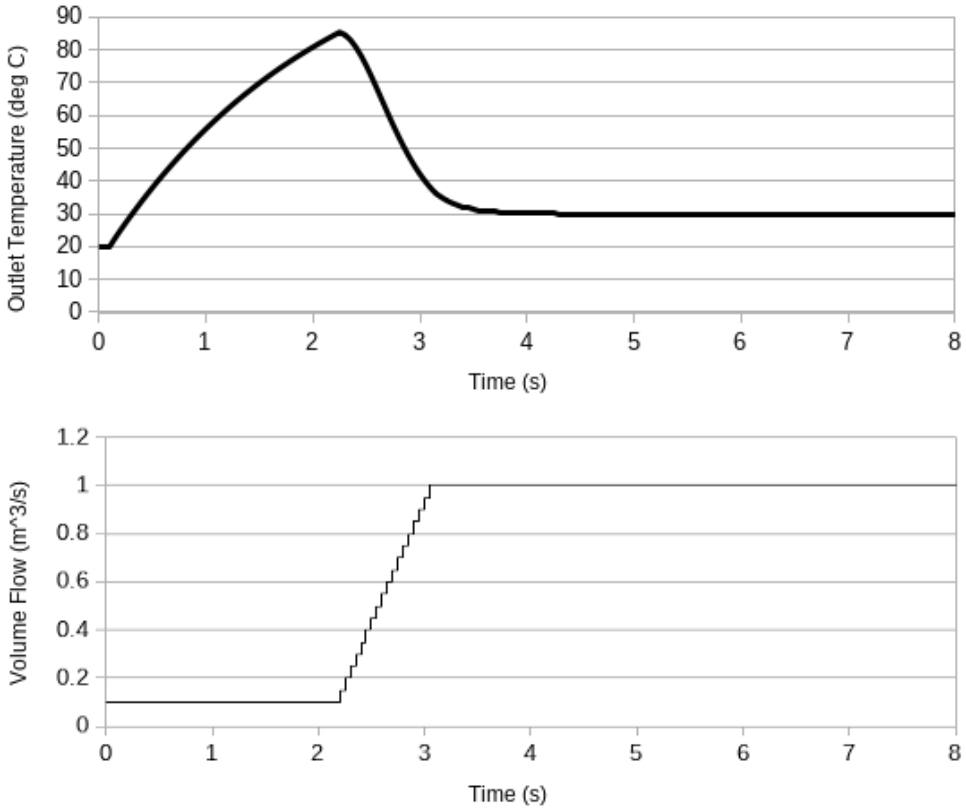

Figure 5.9: Time-Triggered Client Temperature and Volume Flow

Note that due to the small time increments, marks at individual values are not shown, for the readability of the graph. The value of the temperature is shown as continuous (straight connections), as it is assumed to be continuous between the simulation intervals. The volume flow, on the other hand, is shown as a stepped function, and the graph represents the exact values of volume flow at those times. This is because the value of the volume flow was only modified at the simulation intervals, and held constant in between.

Temperature Control Client

The client with automatic time progression represents a temperature control. The control works on approximately 0.05 s time increments. At the start of the increment, the control checks the outlet temperature, if it is higher than a pre-determined trigger value (in this case 40 degrees Celsius), the control starts increasing the volume flow of the pump by 0.01 $\frac{m^3}{s}$ per increment. As long as the temperature is higher than the trigger value, the volume flow of the pump keeps increasing in each increment, up to a cap of 2 $\frac{m^3}{s}$. If the temperature is under the trigger value, the flow of the pump is decreased, by 0.01 $\frac{m^3}{s}$ per increment, minimally down to 0.1 $\frac{m^3}{s}$ (which is also the starting value). The result can

be seen in Figure 5.10.

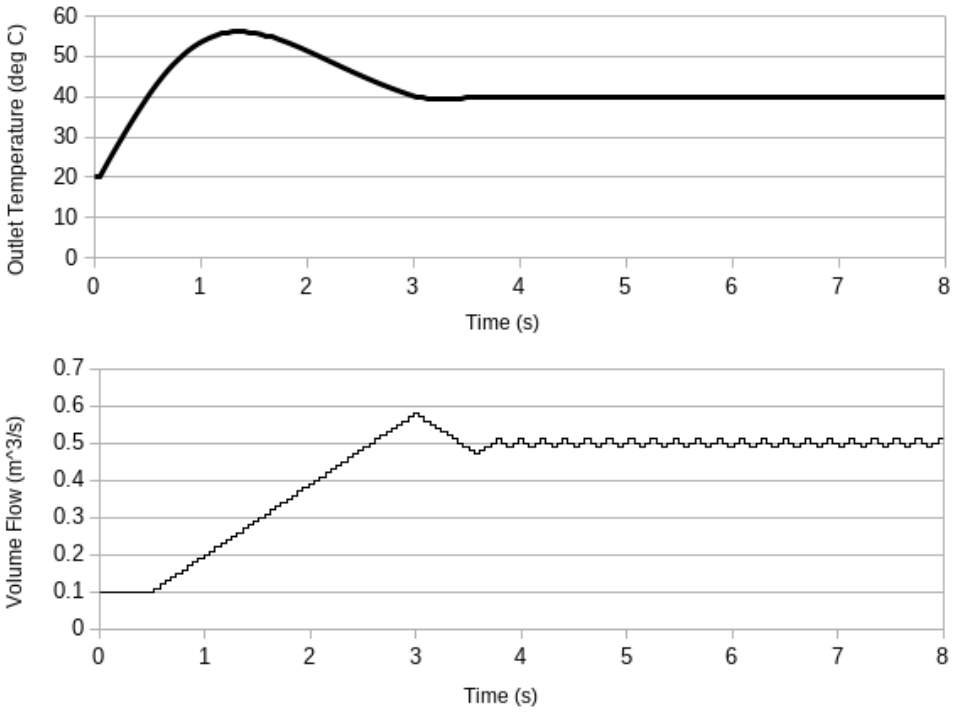

Figure 5.10: Temperature Control Client Temperature and Volume Flow

From the figure it can be seen that the temperature control starts increasing the coolant volume flow as soon as the temperature value rises above 40 degrees Celsius. The volume flow keeps increasing, until the temperature is brought under the trigger value, at which point it starts decreasing again. The slow change speed of the volume flow control is beneficial in this case, and the temperature quickly stabilizes at the trigger value (40 degrees Celsius), with the volume flow around 0.5 $\frac{m^3}{s}$.

CHAPTER 6

Discussion

This chapter presents a discussion of the topics of the thesis, going through the limitations, as well as the possible improvements of the proof of concept implementation, while also discussing difficulties observed while developing the implementation.

Chosen Use Cases

The use cases that were used for the proof of concept implementation were described in Chapter 5.

The first use case presents a relatively simple free fall physics experiment (Section 5.1), which serves to demonstrate the matching capabilities of the proof of concept implementation. The reason for the simplicity is that the number of variables grows rapidly with increasing simulation complexity (using the Modelica simulation environment), thus greatly complicating the matching, as can be seen by the (still relatively simple) second use case (Section 5.2). Even though the physics experiment of the first use case itself is almost trivial (friction-less free fall), there are still 23 variables that are possible matches, on the FMI side. One of the methods in which the matching was made easier for the program, was the development of the information model. Namely, the information model was developed to be a good match for the simulation model. All of the variables have compatible types, matching ranges and engineering units. This greatly simplified the matching for the program, but as can be seen in Subsection 5.1.3, in particular in Table 5.1, the program still needed help from the user via the override file, for one of the variables.

One of the reasons that difficulties can appear is that some variable profiles appear multiple times, for example acceleration appears two times in the accelerate module, and once in the mass module, in the first use case. The matching difficulties can be clearly seen by the second use case, where the program should match three OPC UA variables to three FMI variables, from a pool of 154 (Subsection 5.2.3). In particular the inlet

and outlet temperature variables proved difficult to match, as there are 27 good (level 1) possible matches for them. As the system presents a fluid heat exchange experiment, it is (correctly) expected that there are many temperature variables. This problem is expected only to increase with the increased system size. Knowing which variable out of the 27 good matches is actually the correct match is not a simple task, and requires a good knowledge of the FMI model. On the other hand, the program's suggestions were quite useful for the volume flow variable, and a human operator could easily deduce the match by the variable names, even if they had low knowledge of the models.

Another reason for matching difficulties is the lack of information/criteria. For example, in order to match variables which have neither range nor engineering unit information, one might have to resort to other matching techniques, such as the structural matching, or some additional criteria (as discussed in Subsection 3.2.2). The simulation models for the use cases were generated with OpenModelica, without further (manual) modification. The variables do have some inconsistencies, for example, in the first use case, acceleration has the unit `m/s2`, while the derivative of velocity has the unit `m.s-2`. Despite being identical units in theory, as the proof of concept implementation only treats fully identical units as the same, these two units are treated as different units. Some of the possibilities to handle this issue were given in Subsection 3.2.2.

Matching Criteria

The proof of concept implementation utilizes the following four criteria: the data type, the data range, the engineering units, and the structural matching. And, as was discussed previously, this might not be enough in order to automate the matching process. Some additional matching criteria are presented in Subsection 3.2.2.

In particular, more extensive use of the structural matching could provide many benefits. The current structural matching is only a single level, meaning that only structures with direct children can be matched, and any overarching structures are not taken into account. If multi-leveled matching was introduced, the benefits could be substantial for more complex models. More complex models might contain multiple instances of the same components, which the program could not differentiate alone. However, the human effort would not need to be great, the human simply needs to match structures to each other. The proof of concept can already handle this partly, where if the user matches one of the variables manually in the override file, the program can then recognize which of the identical modules should be matched. In order to facilitate this process and make it easier, the proof of concept implementation would thus need to be expanded so that the structures would be presented separately to the user, and the option would be given given to match structures to each other. A possible use for the naming criterion, which is otherwise difficult to utilize, could be to match these identical structures to each other, if they follow a similar naming convention. For example, if the names were `Fan 1` and `Fan 2` on one side, and `HeatingF_1` and `HeatingF_2` on the other, if the program detects that the structures are the same, it could also detect that there is a part of the name that is the same, and part of the name that is different in each of the naming

conventions (in the example "Fan " is the same for both fans, and the number afterwards differs). If the differing parts match in the two systems they can then be matched to each other. Of course, this relies on the fact that a part of the name is identical, as well as the numbering order of fans (i.e., it might be the case that the fans are in different order in the other system). Utilizing these techniques requires extensive knowledge about the systems, in order to use such speculations with confidence. Even if it would help in this case, the naming criterion is still difficult to use, and might be too speculative for most other use cases.

Experience with the second use case (Section 5.2) has shown that bigger physical systems often contain variables which have similar profiles (for example, temperature variables), and using just the type, range, and unit matching in this case was not enough. The structural matching was also of no use in the particular use case, as the OPC UA data hierarchy did not match the FMI structure (which can also be expected with bigger systems).

A criterion which would greatly help with identifying relevant variables and only matching those is the access level criterion. This would greatly help when the simulations are used to test other parts of the system, as the relevant information for the other parts is only in the input and output variables. The technique would be of no use for the digital-twin simulations, as the internal variables and data are very much relevant in that use case. So, in the case of simulations for tests, if the only required variables are input and output variables, this would greatly reduce the number of variables which need to be matched, as simulations often contain many more internal variables. From the automation system perspective, the access levels would provide the necessary information to determine the input and output variables. For example, the variables that are "read-only" or an equivalent thereof for the automation system client would be the output variables of the simulation. In the simulation system the input and output variables are usually explicitly distinguished by additional information attached to the variable. In the implemented proof of concept for this thesis, this information is extracted from FMI variables, however, as the access levels distinction for the automation system is not implemented, this information is not used for matching. Note that one could still partly utilize this information from the simulation system, even if there is no information on the access levels of the automation system. The simulation variables could be separated into input/output variables, and internal variables. Thus, even if the information which variables are input and which output on the automation side is lacking, we can still extract the relevant variables from the internal parameters, in the simulation system. So, even though access levels would greatly reduce the number of variables on the simulation side that are considered for matching, this criterion can only be used with certain simulation types, namely when we are testing the functionality of the automation system, and cannot be used in simulation types where internal information is desired (digital twin).

In particular, with the second use case of this thesis, the volume flow input variable was designated as input. The inlet and outlet temperature variables were however internal (local) variables. If the FMI model was redesigned in such a way, that these two variables

were output variables, then a matching priority could be given to them. Using this in combination with access levels on the automation system side would probably yield good results. In particular, the volume flow variable would be immediately matched, as there is only one FMI variable designated as input in the five level 1 matches that the program finds for the volume flow. For the temperature there are three variables within the 27 that are already designated as outputs, this means that after this matching criterion would be applied, there would be five level 1 matches for both inlet and outlet temperature variables. Having a list of five possibilities is much more manageable than 27. Even if the user could eliminate the three output variables that are not matches, they would be left with two FMI variables and two OPC UA variables (`ambient1.T_port`, `ambient2.T_port`, `Inlet Temperature`, and `Outlet Temperature`, respectively), without knowing which one matches to which. This would again require the knowledge of the FMI model, in order to make a correct match. In total, the access levels matching criterion could greatly improve the matching for the second use case, automatically matching one of the variables, and offering a smaller possibility set for the other two.

Time Management

The proof of concept implementation offers two modes of time progression: the manual and the automatic time progression. In the manual mode, the time is advanced step-wise by the user, and in the automatic mode, the time progresses naturally. The time modes are described in detail in Subsection 4.2.2. In the program itself, the simulation is advanced step-wise in both of the modes, as the FMI Library provides such interface. The automatic time progression is emulated by executing the time step for the passed time between client calls. There are several possible problems that might occur. First of all, it is not possible for two clients to use different time progression methods simultaneously, as the program only has one simulation running. The first issue with automatic time progression is that there are no latency guarantees. If the client and the server are not on the same machine, the network latency can influence the time of arrival of the data. When testing the system functionality, this might actually be desired. Another issue are simultaneous requests from different clients, which might cause additional latency, from the server this time.

Additionally, there is the issue of long wait times. If the simulation is not advanced in a long time, the time required to calculate the time step might be larger than what the client is willing to wait. For example, there were no calls in the last several hours, and the client now makes two calls, several seconds apart. On the first call, the server needs to calculate the entire simulation progression for the previous several hours, in order to provide the information to the client. This calculation might take considerably more than several seconds after which the second call comes. If the second call is only handled once the calculations are finished, the information provided in the two response to the client might not be only several seconds apart anymore. One possibility is for the client to attach a time-stamp with each of the requests, so the server would know at which time point the information is desired. The problem here is that if there are

multiple clients, they might not be fully time-synchronized, so such approach could lead to problems (for example, a client might request the current information in its own time, but for the server, that time point is already in the past). Alternatively, the server could keep track of the time that the client calls were made using its own clock. This relies on one thread or process handling the client calls, while the simulation is running in another thread or process. Of course, one would then need to handle thread synchronization and communication. A different approach would be to have regular intervals at the server, at which the simulation is advanced, even if no client calls are made. For example, the server advances the simulation every 10 minutes, even if no client requested any data. If these time increments are chosen properly, this method would ensure that the wait times do not get too large. The automatic advancement intervals would need to be chosen strategically or conservatively, as the simulation might not always take the same amount of time to advance. For example, if in a certain interval the physics simulation is fully stationary, the interval might be calculated very quickly, but if there is a force applied to the system, and many physics components start interacting, the simulation time for this interval might be much larger.

Real-time

The issue of real-time should also be mentioned. In this case, real-time refers to processes with guaranteed response time, and not the natural time progression. The time progression in the implemented program, is at no point exact. The simulation results and values do correspond to the time that they are given at on the server-side, however the communication and the response time cannot be guaranteed. The time for the request to arrive to the server is also unknown and not measured in the implementation. This means that the implementation is not real-time capable. The OPC UA was expanded with the publish-subscribe model and with Time-Sensitive Networking (TSN) Ethernet standard. With these improvements, the OPC UA is capable of real-time communication. The FMI for Co-Simulation also supports real-time-capable C code for the FMUs. This means that one could implement a real-time capable version of this proof of concept application; however this is outside of the scope of this thesis.

Requirements

In Subsection 3.2.4, seven requirements were made for the matching process. The adherence of the proof of concept implementation to these requirements will now be discussed. Firstly, the proof of concept matching is based mainly on the given simulation model (FMU) and the information model (OPC UA nodeset file) (RQ1). The matching happens almost fully automatically for some use cases (Use Case I needed only one variable to be matched manually), however there are issues with larger and more realistic use cases (Use Case II), where additional matching criteria might be required (RQ2). The proof of concept program uses engineering units, ranges, and structural information as matching criteria, and the variable type as a disqualifying criteria. The weights are calculated (as described in Subsection 4.2.4) and the matches are given based on those

weights (RQ3). While there are no wrong matches in the given use cases (RQ4), this is not a guarantee with any use case. A simple example which would cause a wrong match would be, in Use Case I, if we wanted the velocity variable of the acceleration module, instead the velocity of the mass module (as was the case in the use case). Based on the structural matching, the program would match the mass module's velocity, as it would have a better value because of the mass variable from the mass module, which is already matched at the time. Although such variable requirements would not make much sense in the particular use case, a similar situation could be created with some other use case, where the matching would make a mistake. Ensuring that there are no wrong matches is not a simple task, and it presents a trade-off, in a sense that: the safer the matching, the less matches there will be, as the program will be less confident in giving out matches. Ultimately, it is up to the human to decide if the matches are correct or not. The program gives the ability to define which OPC UA variables should not be matched (RQ5), as the matching starts from the OPC UA side. When a match for a variable is not found, the program will provide a list of possible matches, with some relevant information (such as data type or engineering unit) (RQ6). The program gives the possibility to manually override or set matches (RQ7). So in total, the requirements RQ2 and RQ5 are partially fulfilled; the only problematic requirement turned out to be RQ4, which did not manifest itself in the use cases, but which could be a problem for other use cases. Solving this requirement is not trivial.

CHAPTER 7

Conclusion and Outlook

Simulations are an important tool when dealing with cyber-physical systems, and as such, they are also part of the ongoing fourth industrial revolution - Industry 4.0. Used either during development, or during deployment, the simulations are versatile and effective, and they help the designers during the whole design process. During development, the simulations are used to test the functionality of a component, by for example, comparing the behavior of the simulation with the behavior of the real physical component. On the other hand, one could test the functionality of other components by letting them interact with the simulation. The simulations can also be used during the deployment phase. The best example for this is the use as a digital twin, where one can monitor the behavior of the component more accurately on the digital twin. The digital twin can also be used for optimization, control, or to diagnose issues.

While the simulations can be used as stand-alone, many benefits can be gained by using them alongside other components, as described above. Interaction with other components requires communication, and this is where issues can occur. The simulations are usually not interoperable by default with the other parts of the automation system. The purpose of this thesis is to explore the possibilities for bringing the simulation and the automation systems together, which is a necessary step for interoperability. In particular, the thesis focuses on the brownfield approach, where the simulation model, and the information model of the automation system are already given, as this is often the case in real life applications, and redesigning these systems is not a possibility. While the greenfield approach (where all system components are designed from the beginning) would allow greater flexibility, and very simplified mapping, it is often not possible to design everything from the start, as the existing systems should be used.

The key component which enables the interoperability is the mapping unit, which stands between the simulation and the automation system, and facilitates communication. In order to do its task, the mapping unit must first match the interface points (variables) of the simulation model, and the information model. This is not an easy task, and

several possible criteria for this matching are explored in the thesis. There are several requirements that such matching should fulfill, and balancing them is not trivial. A trade-off which often occurs in computer science also presented itself here: having a stricter matching means that more found matches will be correct, but it will reduce the number of variables that are matched, as the mapping unit leans on the side of safety. On the other hand, having a more lenient mapping unit might result in more matches, but there might also be more incorrect matches. The two key concepts here are precision and recall. Precision is the number of correct matches within all matches made, and recall is the number of correct matches made, from all correct matches. Ideally we would want that all matches made are correct, and also that we matched all variables that could be matched. In order to lower the probability of incorrect matches, several matching criteria should be and are used, and the possible matches are then ranked according to the weighted criteria which they do or do not fulfill. There is a possibility of human intervention in the matching process, whereby the human can declare that some variables should not be matched (this is useful for automation system variables which have no equivalent nor relevance for the simulation), or match some variables manually, which might be required if the mapping unit cannot deduce all matches on its own. The ease of manual matching is improved by the mapping unit, which provides high probability matches for the human to pick from, with all the required variable meta-data - making the manual matching a straightforward process.

The proof of concept implementation of this thesis utilizes these presented techniques to facilitate interoperability of the simulation and the automation system, demonstrated with two use cases. The frameworks which were chosen for the simulation and for the automation system are FMI and OPC UA, respectively. They were chosen as they are open, standardized, and in widespread use. While the proof of concept can work on other use cases, additional matching criteria might be required for larger use cases, in order to reduce the level of human intervention required.

The mapping unit is presented with the model description file of FMI and the nodeset file of OPC UA. The first use case presents a friction-less free fall. There are four OPC UA variables which need to get matched with their corresponding pairs within 23 FMI variables. This particular use case was chosen as it nicely demonstrates the functionality of the automatic matching as well as the cases where human intervention is required. The proof of concept implementation manages to find the correct matches for three of the four OPC UA variables, utilizing all matching criteria at its disposal. The program is left with a dilemma for the last variable, as there are two good FMI candidates for a match for this variable. Instead of making an incorrect assumption, the program reports its lack of success to the user, and gives them the two good match candidates that it found. Based on the knowledge of the systems and the use case, the user can then provide the last match manually, thus completing the matching, and enabling the data flow between the simulation and the automation system, for all four variables, which was the task.

The second use case demonstrates that larger and more realistic use cases require additional user intervention. This use case presents a fluid heating problem, where a

pump determines the coolant flow to a pipe, which is heated by an external source. The three OPC UA variables should be matched to three FMI variables from a pool of 154. This task is difficult for the proof of concept implementation, and the program does not manage to match any of the three variables. It does, however, provide good suggestions, making the matching procedure for the human easier for two of the variables, and very easy for one. This demonstrates the importance of program's suggestions, despite the lack of success in matching, for the particular use case. The use case also nicely demonstrates the data flow connection, where, in particular one of the clients uses volume flow input and the outlet temperature to dynamically stabilize the temperature at the desired value, by adjusting the volume flow.

The proof of concept implementation also implements two timing modes, intended to facilitate the two use cases: manual and automatic time progression, facilitating development phase and deployment phase simulation uses, respectively. The proof of concept implementation has shown that implementing a mapping unit to connect the simulation and automation systems is possible, and that these systems can be brought to interoperate. It has also shown the importance of choosing good matching criteria, and their weights, which come to a greater effect when applying these techniques to bigger use cases.

A possible improvement for the theoretical part of the thesis would be a closer look at the matching criteria, and in particular the matching weights. While the exact weight values cannot be determined, as they are use-case dependent, it would be possible to consider multiple use cases, and the importance of individual matching criteria, which would then help obtain a general picture on importance of the individual criteria. Another possibility would be to focus on RQ4 which is only partly fulfilled in the current proof of concept. In other words, one should make sure that the program makes no incorrect matches. While complete certainty might be impossible, different weighting techniques and general matching techniques could be used in order to improve the confidence of the matches.

There are two important directions in which expanding the proof of concept implementation of go in. First, one could expand the timing behavior. Some possible issues, and improvement possibilities were already listed, but in particular, one could ensure that the automatic time progression mode is more reactive by automatically advancing the simulation as required. More about this problem can be found earlier in this chapter (under Time Management). The other possible direction would be adding more matching criteria and improving the existing. This would help make more matches, leading to improvements in RQ2. While the present matching criteria were enough for the first use case, larger and more realistic use cases, like the second use case might require additional use of the override mechanic (which is not desirable). Some of the possible criteria were mentioned in Subsection 3.2.2. In particular, the structural matching criterion could be improved to detect multi-layered structures, which would enable even larger sets of variables to be matched at a time. Access level criterion could also help greatly in design phase simulations.

List of Figures

List of Tables

Bibliography

[Bac05] Marko Bacic. On hardware-in-the-loop simulation. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 3194–3198. IEEE, 2005.

[BAS14] Christian Bertsch, Elmar Ahle, and Ulrich Schulmeister. The functional mockup interface-seen from an industrial perspective. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, number 096, pages 27–33. Linköping University Electronic Press, 2014.

[BOA+11] Torsten Blochwitz, Martin Otter, Martin Arnold, Constanze Bausch, Christoph Clauß, Hilding Elmqvist, Andreas Junghanns, Jakob Mauss, Manuel Monteiro, Thomas Neidhold, et al. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference*, pages 105–114. Linköping University Press, 2011.

[BVZ15] EV BITKOM, EV VDMA, and EV ZVEI. *Umsetzungsstrategie Industrie 4.0: Ergebnisbericht der Plattform Industrie 4.0*. Plattform Industrie 4.0, April 2015. Link: `https://www.plattform-i40.de/IP/Redaktion/DE/Downloads/Publikation/umsetzungsstrategie-2015.html` , accessed October 2021.

[GHIU17] Markus Graube, Stephan Hensel, Chris Iatrou, and Leon Urbas. Information models in opc ua and their advantages and disadvantages. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2017.

[JOM16] Václav Jirkovskỳ, Marek Obitko, and Vladimír Mařík. Understanding data heterogeneity in the context of cyber-physical systems integration. *IEEE Transactions on Industrial Informatics*, 13(2):660–667, 2016.

[Led99] Jim A Ledin. Hardware-in-the-loop simulation. *Embedded Systems Programming*, 12:42–62, 1999.

[Li18] Ling Li. China's manufacturing locus in 2025: With a comparison of "made-in-china 2025" and "industry 4.0". *Technological Forecasting and Social Change*, 135:66–74, 2018.

[LKYO17] Byunghun Lee, Dae-Kyoo Kim, Hyosik Yang, and Sungsoo Oh. Model transformation between opc ua and uml. *Computer Standards & Interfaces*, 50:236–250, 2017.

[LM06] Stefan-Helmut Leitner and Wolfgang Mahnke. Opc ua–service-oriented architecture for industrial applications. *ABB Corporate Research Center*, 48(61-66):22, 2006.

[LSS19] Hasan Latif, Guodong Shao, and Binil Starly. Integrating a dynamic simulator and advanced process control using the opc-ua standard. *Procedia Manufacturing*, 34:813–819, 2019.

[Lu17] Yang Lu. Industry 4.0: A survey on technologies, applications and open research issues. *Journal of industrial information integration*, 6:1–10, 2017.

[MGGU11] Wolfgang Mahnke, Andreas Gössling, Markus Graube, and Leon Urbas. Information modeling for middleware in automation. In *ETFA2011*, pages 1–7. IEEE, 2011.

[MLD09] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC unified architecture*. Springer Science & Business Media, 2009.

[Mod20] Modelica Association. *Functional Mock-up Interface for Model Exchange and Co-Simulation, Version 2.0.2*, December 2020. Link: `https://fmi-standard.org/downloads/` , accessed April 2021.

[NFCM19] Elisa Negri, Luca Fumagalli, Chiara Cimino, and Marco Macchi. Fmu-supported simulation for cps digital twin. *Procedia manufacturing*, 28:201–206, 2019.

[OPC20] OPC Foundation. *OPC Unified Architecture, Release 1.04*, 2017-2020. OPC 10000-1/14 ; Link: `https://opcfoundation.org/about/opc-technologies/opc-ua/` , accessed October 2021.

[RMK16] Vasja Roblek, Maja Meško, and Alojz Krapež. A complex view of industry 4.0. *Sage Open*, 6(2):2158244016653987, 2016.

[RR20] Jan Reitz and Jürgen Roßmann. Automatic integration of simulated systems into opc ua networks. In *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, pages 697–702. IEEE, 2020.

[SFT+19] Diana Strutzenberger, Thomas Frühwirth, Thomas Trautner, Ronald Hinterbichler, and Florian Pauker. Communication interface specification in opc ua. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1329–1332. IEEE, 2019.

[SSK+20] Gernot Steindl, Martin Stagl, Lukas Kasper, Wolfgang Kastner, and Rene Hofmann. Generic digital twin architecture for industrial energy systems. *Applied Sciences*, 10(24):8903, 2020.

[TS16] Lane Thames and Dirk Schaefer. Software-defined cloud manufacturing for industry 4.0. *Procedia cirp*, 52:12–17, 2016.

[WMO+16] Stephan Weyer, Torben Meyer, Moritz Ohmer, Dominic Gorecky, and Detlef Zühlke. Future modeling and simulation of cps-based factories: an example from the automotive industry. *Ifac-Papersonline*, 49(31):97–102, 2016.

[YJR+10] Yong-Ho Yoo, Thomas Jung, Malte Roemmermann, Malte Rast, Frank Kirchner, Jürgen Roßmann, and Robotics Innovation Center. Developing a virtual environment for extraterrestrial legged robot with focus on lunar crater exploration. In *Proceeding of 10th International Symposium on Artificial Intelligent, Robotics and Automation in Space*, volume 29, 2010.

Appendix

The Appendix can be understood as an extension of Subsection 4.2.4 (in particular the "Applied Matching Techniques" segment). Here, the exact matching weights are presented for the matching algorithm, based on the matching criteria (see Subsection 3.2.2 for the list of possible criteria). These values were used in the proof of concept implementation (Chapter 4).

Keep in mind that the values for the weight in the proof of concept implementation are as follows: 0 means that the match is not possible, 1 means that the match is highly likely, 2 means that the match is slightly less likely, and so on, each higher value indicating that the match is less likely; apart from 0 which is a special value meaning that the match is not possible. In the proof of concept implementation, weights up to 3 are taken into account for matching, all values higher than that are treated as too unlikely for a match, and are not considered. Overrides are handled separately, they exclude the variables from the matching procedure, and the override pair is taken as a certain match (if possible). A variable pair needs to pass the Data Type criterion (disqualifying criterion), in order to be able to be matched.

Data Type Criterion

The first table (Table A) determines the starting values, based on the data type. Data type is a disqualifying criterion, meaning that a mismatch will result in "Match not possible". "Not possible" in the table refers to weight 0. The table shows only a subset of OPC UA variables; these variables are recognized by the proof of concept program.

Table A: Table of starting weights, based on data types

FMI Type	OPC UA Type	Starting weight
Real	Float	weight = 1
	Double	
	Duration	
	Otherwise	Not possible
Integer	SByte	weight = 1
	Int16	
	Int32	
	Byte	
	UInt16	
	UInt32	
	Otherwise	Not possible
Boolean	Boolean	weight = 1
	Otherwise	Not possible
String	String	weight = 1
	LocaleId	
	QualifiedName	
	GUID	
	LocalizedText	
	ByteString	
	DiagnosticInfo	
	XMLElement	
	ImageBMP	
	ImageJPG	
	ImageGIF	
	ImagePNG	
	Otherwise	Not possible
Enumeration	UInt32	weight = 1
	Otherwise	Not possible

In order to be processed further, a variable pair needs to be possible to match. This means that penalty application is not applied if the match was not possible (as determined by the data types, from Table A).

Data Range Criterion

The second table (Table B) presents the matching penalties based on value ranges. "FMI DR Max" value is the maximum data range limitation of the FMI variable ("FMI DR Min" is the minimum), if present. If the value is not present, the condition is treated as not fulfilled (and the penalty is not applied). So, for example in the expression "FMI DR Max $\leq$ FLOAT_MAX", the condition is fulfilled if FMI DR Max exists, and is smaller or

equal to the maximal float value. Penalty is applied only once per table. So if a variable fulfills multiple entries in a table, it is penalized only once.

Table B: Table of penalties based on value ranges

FMI Type	OPC UA Type	Condition	Penalty
Real	Double	FMI DR Max $\leq$ FLOAT_MAX	weight = 2
	Duration		
Integer	SByte	FMI DR Min ≥ 0	
	Int16	FMI DR Min ≥ 0	
		FMI DR Max $\leq$ SCHAR_MAX	
	Int32	FMI DR Min ≥ 0	
		FMI DR Max $\leq$ SHRT_MAX	
	UInt16	FMI DR Max $\leq$ UCHAR_MAX	
	UInt32	FMI DR Max $\leq$ USHRT_MAX	

Additionally, apart from the possible penalties in the table, a penalty is applied if both FMI and OPC UA variables in a pair have their minimum values defined, but the minimum values do not match. Same is true for the maximum values. The penalty is again equal to 1, but the total weight for this step cannot exceed 2. This means that no matter how many data range tests a certain variable pair fails (including those in the Table B), the maximum penalty for data range criterion is still 1. No penalty is applied if a minimum or a maximum value is defined on one side, but not on the other. This is because such definitions might be necessary, as the data types are not equal on both sides. For example, an FMI Integer might have a minimum value of 0, and match to an UInt16 on the OPC UA side. In this case, the data range limit (of minimum value being 0) is not required on the OPC UA side, as it is implied by the data type. Penalizing the variable pair in this case would be incorrect.

Engineering Unit Criterion

For engineering units, a penalty of 1 is applied if either of the following is true:

- If both FMI and OPC UA variable have their engineering units defined, but the engineering units are not the same (string comparison).
- If one variable has the engineering unit defined, but the other one does not.

As can be seen from the second item, the penalty is also applied when only one of the engineering units is defined. This is done because not having an engineering unit can be a designation on its own. No penalty is applied if both variables have a single engineering unit (same strings by both variables), or if both variables lack an engineering unit.

Structure Criterion

For the structural criterion, after a match is made (either by the matching algorithm or through overrides), variable structures are inspected. In FMI this refers to the lowest structure (which is designated by the name of the entire structure chain) containing the variable. For example: in `robot.arm1.centerOfMass`, the structure of the variable `centerOfMass` is `robot.arm1`. In OPC UA structure refers to the (node id of the) direct parent of the node.

Let's say that FMI variable F and OPC UA variable O are a match. Their structures are: $F.s$ and $O.s$. To perform the structural matching, the program then iterates through all combinations of FMI and OPC UA variables (f, o), and applies the penalty of 1 in the following cases:

- If $F.s = f.s \wedge O.s \neq o.s$
- If $F.s \neq f.s \wedge O.s = o.s$

Meaning that the penalty of 1 is applied one of the structures in the iterating variable pair is the same as in the matched pair, but the other one isn't. If both of the structures are the same in the iterating pair as in the matched pair, or if neither of the structures are the same in the iterating pair as in the matched pair, no penalty is applied.